CIRCUMVENTION OF CENSORSHIP OF INTERNET ACCESS AND PUBLICATION

BY

FREDERICK ERNEST DOUGLAS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Associate Professor Matthew Caesar, Chair
Associate Professor Nikita Borisov
Professor Klara Nahrstedt
Assistant Professor Eric Wustrow, University of Colorado, Boulder

ABSTRACT

Internet censorship of one form or another affects on the order of half of all internet users. Previous work has studied this censorship, and proposed techniques for circumventing it, ranging from making proxy servers available to censored users, to tunneling internet connections through services such as voice or video chat, to embedding censorship circumvention in cloud platforms' front-end servers or even in ISP's routers.

This dissertation describes a set of techniques for circumventing internet censorship building on and surpassing prior efforts. As is always the case, there are tradeoffs to be made: some of this work emphasizes deployability, and some aims for unstoppable circumvention with the assumption of significant resources. However, the latter techniques are not merely academic thought experiments: this dissertation also describes the experience of successfully deploying such a technique, which served tens of thousands of users.

While the solid majority of previous work, as well as much of the work presented here, is focused on governments blocking access to sites and services hosted outside of their country, the rise of social media has created a new form of internet censorship. A country may block a social media platform, but have its own domestic version, on which it tightly controls what can be said. This dissertation describes a system for enabling users of such a platform to monitor for post deletions, and distribute the contents to other users.

To my wife, my parents, and my cats Oolong and Lilly,
for their love and support.

ACKNOWLEDGMENTS

I would like to acknowledge the people whom I worked with on these various projects: Nikita Borisov, Matthew Caesar, Daniel Ellard, Sergey Frolov, Alex Halderman, Christine Jones, Allison McDonald, Rorshach[1], Steve Schultze, Will Scott, Weiyang Pan, Ben VanderSloot, and (last but not least!) Eric Wustrow.

Additionally, I would like to thank the developers of the SoftEther VPN system, which the Salmon implementation is built upon, for releasing it under the GPL. I thank ntop for providing the deployable TapDance project with academic licenses for the zero-copy version of PF_RING.

[1] Pseudonym.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Governments have always sought some degree of control over what their citizens say, regardless of the means of communication involved. Censorship has followed civilization onto the internet, with governments blocking traffic to undesirable web sites and services based in freer countries. Data from Freedom House indicates that on the order of half of the world's internet users live under some significant form of internet censorship (illustrated in Figure 1.1).

Internet censorship includes simple techniques such as disruption of DNS resolutions of banned domain names [1] and blocking of flows to blacklisted IP addresses, as well as more sophisticated techniques, such as killing flows when deep packet inspection detects a banned keyword [2], identifying Tor connections with traffic pattern fingerprinting [3, 4], and (not yet widely deployed, but practical) general fingerprinting of websites' traffic patterns within encrypted tunnels [5, 6].

Anti-censorship techniques should defeat any such censorship technique (with the possible exception of traffic fingerprinting, which is not yet a real issue and can typically be dealt with in a separate component [7]). The simplest technique is to connect (with encryption) to a proxy server: a server that performs internet communication on behalf of a client, making the client's internet location appear to be the proxy's address. A user with access to a proxy server has escaped the censor: the encrypted data being transferred between the user and proxy is not visible to the censor, and no censor currently blocks encrypted communication entirely. However, the proxy approach relies on the censor not realizing the proxy is a proxy, at which point its IP address can simply be added to a block list. Any system employing proxy servers is attempting to reconcile the contradictory goals of widespread distribution of proxy serves, and keeping the identities of the proxy servers hidden from the censor.

Sites and services hosted within a censored country can be compelled to remove offending content. A censor seeking to block websites is really seeking to prevent the inward flow of unwanted ideas from other countries. However, blocking outside information is only half of the censor's task.

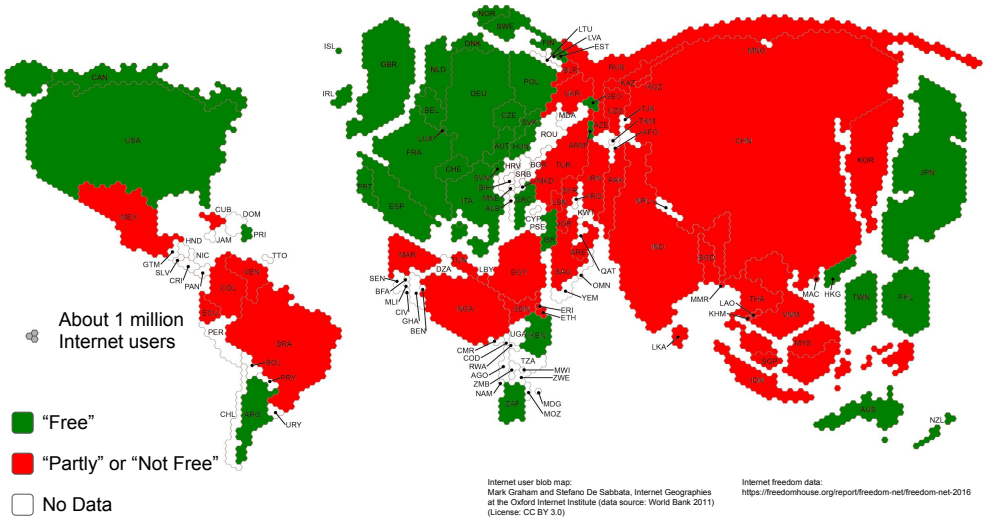

Figure 1.1: Freedom House's overview [8] of internet freedom, overlaid on a map of geographical distribution of internet users.

Effective censorship must also prevent people from expressing their own ideas. Social media represents a unique challenge to a censor aiming for this goal. At a traditional newspaper, a handful of known employees write articles that can be carefully monitored, perhaps even requiring explicit approval from the censor to publish. With social media, a significant fraction of the country's entire population is speaking its mind, spontaneously and constantly. Repressive governments, China in particular, attempt to control social media as tightly as they can, deleting undesirable posts.

This dissertation presents work intended to enable the circumvention of both types of censorship: censorship preventing people from accessing external information (access censorship), and censorship preventing people from expressing themselves (publication censorship).

1.1 Access Censorship

We can divide efforts to defeat access censorship into two categories: those in which circumvention resources are easily blocked, and must be kept hidden from the censor, and those in which the circumvention is inextricably tied to a service that the censor is unwilling to block. This dissertation will describe a new technique of the former class: Salmon [9], which intelligently distributes proxy servers. It will also describe a new technique of the latter class: DittoTap, an advanced form of decoy routing (where censorship circumvention happens within ISPs' routers). In addition to DittoTap, it will describe a successful effort to deploy a production-quality implementation of an existing decoy routing technique, TapDance[1] [10], in the real world with tens of thousands of users (and thousands of concurrent users).

Salmon has the advantage of being incrementally deployable, even by individuals on home internet connections. The same is generally true of other easily-blocked-helper techniques, as well: the easily blocked helpers are typically envisioned as individually volunteered proxy servers (or Tor hidden bridges).

TapDance and DittoTap require the support of large ISPs, but draw close to the ultimate goal of censorship circumvention: a system that is freely available to the general public, provide high bandwidth and low latency, and cannot be blocked without severing internet connectivity to the outside world.

1.2 Publication Censorship

Social media services are massively popular, and in particular have become a significant component of political discourse. If an educated population with exposure to the internet were denied access to social media platforms, the impetus to seek out effective circumvention techniques would increase dramatically. Therefore, an intelligent censoring government will provide its citizens with functional, well-designed social media platforms—that are entirely under its control. In this way, the government can prevent online

[1]This dissertation describes the development and large-scale deployment of a production-quality TapDance implementation. I was not involved in the original conceptual design or publication of TapDance.

communities of dissidents from forming, without angering its populace with the obviously and tangibly oppressive move of disallowing social media.

With the perfectly free internet access provided by Salmon, or DittoTap, or other systems, people unhappy with the censored social media platforms available to them could in theory simply switch to an uncensored social media platform. However, network effects are powerful. Especially because not all users care about censorship, expecting the bulk of (for example) Sina Weibo users to use anti-censorship tools to switch to Twitter is unrealistic. The best way to support free discussion is to attack censorship on platforms that are already in common use. GhostPost [11], addresses this need by supplying its users with "resurrected" copies of deleted posts, focusing its resources on protecting exactly the posts that its users are interested in.

1.3 Structure

This dissertation is structured chronologically, starting from Salmon. The ordering of the work is important to keep in mind: a central motivation for the Salmon project was the apparent infeasibility of decoy routing, based on the fact that the idea had been around for five years without a deployment beyond lab-scale. In theory, ignoring issues of feasibility, decoy routing has always been the most promising anti-access-censorship approach. Now that decoy routing has been demonstrated to be feasible, efforts to fight access censorship should focus on this approach.

The chapters are structured in this way:

- Chapter 2 surveys relevant previous work on internet censorship (both of access and publication).
- Chapter 3 describes Salmon, an algorithm for intelligent, careful distribution of proxy servers for circumvention of access censorship, based on separating users by the amount of trust they appear to be due, as well as a full implementation and trial deployment of that system.
- Chapter 4 describes GhostPost, a system for the distributed preservation and restoration of users' posts on social media platforms under censorship. In particular, this section describes an implementation and trial deployment aimed at Sina Weibo (`weibo.com`).

- Chapter 5 describes the recent deployment of a production-quality TapDance implementation, which served tens of thousands of users (up to around 4,000 concurrently).

- Chapter 6 describes an addition to the TapDance protocol that fixes vanilla TapDance's extreme constraint on upload throughput.

- Chapter 7 describes the theory, prototype implementation, and evaluation results of DittoTap, a new decoy routing design that combines TapDance's passive operation with Slitheen's perfect imitation of decoy site traffic characteristics.

In general, the common thread running through these works is deployability. Before decoy routing was recently demonstrated to actually have a path to deployment, Salmon was an effort to develop the most robust and algorithmically rigorous system possible while still being fully deployable. In particular, Salmon aimed to provide the same service (building on the same software) as the already deployed VPN Gate [12], but with an evaluated algorithm handling server distribution.

GhostPost included a Chrome extension built to not interfere with users' `weibo.com` use in any way, seamlessly inserting deleted posts where they would have been had they not been deleted. Additionally, GhostPost is incrementally deployable; even in a deployment with a single GhostPost user, that user's GhostPost process will preserve some deleted posts they otherwise would have missed.

The TapDance deployment project obviously focused on deployment, and the work done to lift its upload cap is for the purpose of making the deployment more usable.

Finally, while DittoTap is still primarily a conceptual design, its central goal is deployability: just as TapDance's motivation was to make a version of decoy routing palatable to ISPs, DittoTap's goal is to make a version of the Slitheen design that is similarly palatable. Additionally, its prototype is built on top of the deployed TapDance station, with nearly no changes to the underlying logic.

Both chapter 3 and chapter 4 (Salmon and GhostPost) are modified and expanded versions of previously published papers [9,11]. Both were published in venues (PETS and USENIX FOCI, respectively) that are open-access, and leave copyright ownership with the author: no permission is necessary to use this material here.

CHAPTER 2

RELATED WORK

There is existing work on both the censorship of access to information, and the censorship of publication of information, as defined in the previous section. Most of the work is on the former.

2.1 Censorship of Access

Some anti-censorship systems force a censor to choose between entirely blocking a legitimate service their citizens find useful, and accepting that their censorship can be evaded. Ideally, such an approach should force the censor to do no less than blocking the entirety of the internet outside the country's borders. Other circumvention techniques rely on helpers that can be blocked by the censor with no collateral damage, and so are effective only as long as those helpers stay hidden from the censor.

2.1.1 Easily Blocked Helpers

VPN Gate [12] hinders a censor's automated processes for large-scale blocking of VPN servers. VPN Gate publishes (in chunks of 100) a complete list of the servers in its system. However, it also mixes in fake IP addresses, and addresses of real web sites the censor does not want to block. To stymie the obvious solution of testing all the addresses for the presence of a functioning VPN server with an automated scan, the proxy servers coordinate to detect and ignore patterns of connection attempts that appear to come from an automated scanning process.

The system as implemented is not actually immune to simple automated probes. A simple harvesting script, run for 9 days on a single free Amazon EC2 instance, was able to verify that 3,101 of 44,039 IP addresses are work-

ing VPN Gate servers. The script evaded the collective defense mechanism while harvesting from just a single IP address by exploiting VPN Gate's *raison d'tre*: after verifying a VPN server by successfully connecting to it, the scraper used that connection to make its next query for new servers.

With that list of the 3,101 working servers, a censor could, with no collateral damage, essentially completely shut down VPN Gate. Only a handful of unreliable servers would remain available, hidden among many blocked and fake servers.

Tor: Tor's [13] goal of internet *privacy* is orthogonal to our goal of internet *access*. However, the Tor Project has also attempted to provide access to censored users by distributing hidden "bridge" relays, but without a rigorous method for pinpointing infiltration by a censor. Designs using Tor do so only to gain Tor's privacy; their ability to provide access would not be reduced by switching to one-hop proxies.

Careful proxy distribution: Some techniques have been developed that aim to prevent a censor from blending in with users, and discovering all of the proxies. Salmon falls under this category. These proposals include recent work such as Proximax [14] and rBridge [15], earlier work such as keyspace hopping [16], and an analysis of Tor blocking [17] containing a sketch of a proto-Salmon or -rBridge. These techniques differ from Salmon, and from each other, both in the algorithms they use to identify censor agents, and in how they attempt to limit the number of agent users the censor can register.

Keyspace hopping [16] is completely open to new users; a user can always request more servers, and users are never banned. To slow the censor, users' requests for servers are limited by computational puzzles. We do not believe this is a feasible approach, as it assumes that *e.g.* the Chinese government cannot computationally dwarf the home computers of all Chinese internet users seeking to evade censorship. Because the system is built around rate-limiting user identities, rather than banning them for bad behavior, a robust system cannot be obtained by simply replacing the repeatable computational puzzles with Salmon's one-time social network registration.

Proximax [14] is purely invitation-based, with an approach even more open than Salmon's rate-limited recommendations. In fact, "invitation" is not quite the right term. The vast majority of users are not official Proximax users; friends simply share any and all servers they discover. The system compensates for this openness by being much more restrictive with

distribution of new servers. Only the very small group of officially registered users, who are roughly equivalent to Salmon's special above-the-trust-system users, are given fresh servers. They are free to distribute them as they wish, but Proximax tracks which servers given to which registered users become blocked, and prefers to give servers to the registered users whose friends get the fewest servers blocked.

rBridge, discussed next, was demonstrated to resist a censor's attack somewhat more effectively than Proximax. Proximax's less accurate discernment makes intuitive sense: it operates at a coarser resolution. Proximax tracks behavior at the granularity of entire social graph clusters, whereas Salmon and rBridge examine individual users.

rBridge: Like Salmon, rBridge [15] seeks to distribute proxy servers (Tor bridges, specifically) to the public while limiting a censor's ability to block them. As its core feature, rBridge prevents the central directory server from learning which users received which servers, in order to protect users from an attack in the context of Tor. Users receive a stream of credits while their servers remain unblocked, and can redeem credits to receive more servers. Users can only join via recommendation, and existing users are only eligible to obtain recommendation tickets if they keep their credit balance above a threshold. Salmon offers major improvements over rBridge in a few regards: robustness, accessibility for new users, and efficient use of server resources.

First and most simply, Salmon is far more robust to an attacking censor than rBridge. Salmon's trust levels, combined with permanent banning, make it over three times as robust to the sort of attack that rBridge considers, as shown in Salmon's evaluation section, subsection 3.4.5.

Second, rBridge does not offer a particularly friendly, or even predictable, path to becoming a user. rBridge only accepts new users who have gotten an invitation ticket from a friend, which the directory server distributes to random users who have kept their credit balance above a threshold. We seek to provide access to all censored users, not just friends of trusted users. rBridge users have no way to know when (if ever) a ticket might be issued to them. They have no control over the process, beyond requesting few enough bridges to remain eligible to receive a ticket.

Third, by designing specifically for Tor, rBridge accepts significantly degraded performance for any given amount of server resources. Tor's multi-hop design results in a third of the aggregate bandwidth and twice the RTT (for

Tor's default of 3 hops) of a one-hop system with equivalent server resources. VPN Gate cites performance as a reason for providing VPN servers rather than Tor bridges. Because our first concern is providing high performance circumvention, we also prefer the one-hop design.

In theory, Salmon, rBridge, and any other careful distribution approach can distribute either Tor bridges or VPN servers. The choice is not tied to the details of the distribution strategy, but rather to implementation goals. That said, rBridge's tradeoffs and distinguishing features would be wasted outside the context of Tor.

Flash proxies [18] are an especially creative attack on the censor's ability to block proxies. Rather than hiding the servers, this approach aims to constantly add new ones to the system, so quickly that the censor simply cannot keep up. Participating websites embed scripts that turn visiting web browsers into Tor bridges for the duration of their visit to the page. While not too harmful to the visitor and certainly for a good cause, this behavior could result in backlash against the hosting sites, and the overall flash proxy concept. More importantly, the inherently short-lived nature of flash proxies make them unsuitable for many popular services, such as video chat, streaming video, and online gaming.

Paid services: There is a thriving marketplace for VPN servers, and money is not a bad way to rate limit the censor. However, we would like to give all internet users access to the entire internet, regardless of their ability to pay for it. Furthermore, the "ability to pay" goes beyond financial resources: how does an otherwise law-abiding citizen pay for an illegal online service provided by a foreign company? Even the pseudonymity of Bitcoin is not (safely) available to Chinese users, as the Chinese government has outlawed Bitcoin. Chinese and Iranian users are currently able to make these payments, but as with VPN Gate, this is the result of a censor that is (for the time being) content to put up just enough obstacles to circumvention that only the most persistent users will bother.

The opposite approach is also possible: rather than have clients pay for circumvention services, have popular censored *destinations* pay. This approach is taken by the popular app Psiphon, with entities such as the British Broadcasting Corporation paying for high-performance delivery of their sites to users under censorship.

2.1.2 Hard to Block Helpers

Decoy routing: Radically different from proxy server approaches, decoy routing seeks to embed censorship circumvention into the fabric of the internet. This concept was independently proposed as decoy routing [19], Cirripede [20] and Telex [21]. TapDance [10] is an evolution of Telex, which does not interfere with packet forwarding at all, in order to be more palatable to reluctant ISPs. Rebound [22] is another evolution, still interfering with packet forwarding, but blending in with normal traffic (at very low throughput) much better than the prior generation, by causing the decoys themselves to deliver covert data to the user inside specifically provoked HTTP error pages.

The general decoy routing approach is for the client to hide a special value in a field meant to be random (TCP ISNs for Cirripede, TLS nonces for Telex, TLS ciphertext for TapDance) to complete a key exchange and indicate their desire to access censored sites. Routers throughout the internet are modified to monitor for and handle such flows. The entire exchange is (at first glance) invisible to the censor: the client initiates the connection to a site that is allowed by the censor, and known to use TLS. Until the traffic reaches the decoy router, the packets' bytes are indistinguishable from traffic truly meant for the overt site. However, an extremely attentive censor may be able to differentiate based on any deviation from the overt site's behavior, in terms of packet size, timing, TCP options, etc. Slitheen [23] solves this problem by injecting data into the husks of actual packets from the overt site.

It is important to consider whether the client would even be able to reach a decoy router in a realistic deployment. It appears decoy routing can withstand [24] the best proposed attacks [25] of that kind.

Domain fronting: [26] In addition to decoy routing's strategy of redirecting traffic at collaborating nodes in the network layer of the internet, it is possible to do similar redirections further up the stack. CDNs can function by having browsers address HTTP(S) requests to the true logical destination (the domain name in the "Host" field of the HTTP header), but physically send them to an IP address controlled by the CDN. If that CDN server does not have the requested item, it can retrieve it from the true server, and then finish the client's request. If the HTTP Host field is hidden inside a TLS session, and the visible portion of the request (such as DNS resolution) pretends to be aimed at an innocuous site also hosted on that CDN, the censor

cannot distinguish requests to blocked sites from requests to innocuous sites.

Domain fronting is an interesting counterpart to decoy routing, being (from a very high level view) the exact same technique applied at a different networking layer. Its use of CDNs and cloud hosting, rather than the single-purpose and relatively inaccessible world of the internet's connective fabric, would appear to make it more easily deployable than decoy routing. However, from anecdotal experience, some cloud providers are not willing to host a domain fronting deployment due to previous bad experiences. Outside of the issue of deployability, decoy routing definitely has an edge: domain fronting's collateral damage is limited to the other sites on the cloud host(s), whereas with domain routing it is potentially as much as the entire internet outside of the censored country.

Piggybacking: Anything that sends encrypted and authenticated messages to any recipient in the world, and is allowed by the censor, can naturally be used to break censorship. There have been proposals to tunnel IP packets through services such as secure VoIP calls [27], video calls [28], and email [29]. Tunneling only through VoIP does not meet our bandwidth goals, as it is essentially dial-up (or worse, given the aggressive psycho-acoustic compression algorithms used in modern voice communication). Worse, because systems such as Skype have no reason to recover dropped packets, the censor can attack covert tunnelled flows with tolerable levels of disruption to Skype calls [30]. Furthermore, secure VoIP and video chat do not enjoy the status of being painful for a censor to block. In fact, these protocols are often targets themselves, for the same reasons that censors block social media platforms.

2.2 Censorship of Publication

The issue of censorship of speech on social media platforms, being an issue that has only come into existence in the past decade or so, has received less attention than censorship of internet access. Several measurement studies have recently been published; these are valuable for obtaining a sense of exactly what a system like GhostPost is up against. These studies gather data on various features of censorship, such as how long posts survive before being deleted, what keywords are associated with censorship, which accounts are most heavily censored, the geographic distribution of censorship, etc.

One study [31] examined the lengths of time that posts ultimately fated to be deleted survive for on Sina Weibo. The authors observed a few interesting features in the deleted posts they studied. First, there is a significant diurnal cycle to deletion times: depending on the time of day when a deletion-fated post is posted, median time to deletion ranges from around 2 to 9 hours. Second, posts survive for a wide range of times: although nearly 90% of deletions happen within 24 hours, some take over a month. Third, there appears to be a moderate negative correlation between number of posts an author has had censored, and the average lifetime of their deleted posts; they hypothesize that successive deletions lead to closer scrutiny from the censor. Among the measurement studies, this one is the most important to GhostPost; the information it provides allows our simulations of a large scale GhostPost deployment to be based on observed reality.

It is also useful to have a sense of what topics are targeted for censorship, who Weibo users are, and how they act. Another study [32] sought to compile a list of keywords most associated with censored posts. The same research group published another study [33] examining the properties of Weibo's user-base.

In order to carry out the study of censored keywords ([32]), the authors built a tool they call Weiboscope. Weiboscope monitors Weibo accounts with at least 1,000 followers, watching for disappearing posts. Weiboscope's collection of deleted posts is accessible through a website, which presents statistics on Weibo censorship, and allows its corpus of deleted posts to be searched by keyword.

In addition to Weiboscope, FreeWeibo (run by greatfire.org [34]) also preserves deleted posts for public viewing. These services both use a centralized scraping system to monitor Weibo accounts deemed most important, and make a database of these accounts' deleted posts available for viewing.

Preserving posts in case they are deleted is not the only way to fight the Weibo censors. For instance, Chinese social media users have for some time been employing homophones, puns, and other linguistic tricks that are much more transparent to humans than to automated processes. Interfering with the censor's automated tools makes their job more difficult, thereby keeping posts alive longer. A recently published approach [35] seeks to automate this process, by developing a Chinese homophone substitution algorithm to be applied to Weibo posts.

2.3 Summary

Access-censorship (blocking of websites) circumvention has seen numerous novel academic proposals, and a few major deployments (academic and otherwise). Decoy routing, the most interesting class of academic proposals, has not yet been seriously deployed, although work described in chapter 5 demonstrated a large-scale (tens of thousands of users) decoy routing deployment that lasted for about a month. Until decoy routing deployments become more commonplace, the deployed state-of-the-art is domain fronting, which is used (as one of many techniques) by the popular app Psiphon. VPN Gate and Tor's hidden bridges are both active, fairly large-scale instantiations of the concept of networks of volunteer proxy servers, although neither is as effective in safely and efficiently distributing servers as it could be.

Publication-censorship (censorship of content posted on a censor-controlled website by users) circumvention, on the other hand, has seen little work beyond measurement and monitoring. The most interesting academic proposal for directly fighting it is to automate a technique that humans are already using: linguistic tricks to confuse censors' automated tools without losing the human meaning of a post.

CHAPTER 3

SALMON: ROBUST DISTRIBUTION OF PROXIES

A proxy server—a server that forwards traffic for a client connected to it by an encrypted tunnel—is a straightforward way for an individual user to escape a censor. Distributing a large number of proxy servers (mostly provided by individual volunteers) is therefore a (seemingly) straightforward approach to fighting censorship. However, nothing stops the censor from posing as a legitimate user: even if the system can somehow enforce a real identity policy, the censor's employees are citizens of a censored country, just like the legitimate users. If there are no countermeasures in place, both in the form of limits on how many identities the censor can create and how much damage the censor can do with each of its identities, the censor will be able to discover and thus block many or all of the system's proxy servers. Proxy-server-based approaches therefore revolve around the careful distribution of the precious resource of proxy server IP addresses.

In addition to several academic proposals [14–16], there are two notable deployments of proxy-based censorship circumvention: VPN Gate [12] and Tor's [13] hidden bridges. Both of these systems manage proxy distribution with *ad hoc* methods, resulting in implementation-based cat-and-mouse games with the censor.

Salmon [9] is an attempt to build a system in the same deployable fashion as VPN Gate (in particular, using the VPN system SoftEther, which is the VPN Gate authors' primary project), but with carefully designed algorithmic logic backing the distribution strategy. Salmon is both an algorithm for distributing a scarce resource (VPN servers) in the face of an adversary trying to destroy that resource (the censor with its ability to block), as well as the name of a deployed proxy server network that employs the Salmon algorithm for distribution.

3.1 Problem and Design Overview

The Salmon algorithm has three core components. 1) It tracks the probability that each user is an agent of the censor (*suspicion*), and ban likely agents. 2) It tracks how much *trust* each user has earned, and distributes higher-quality servers accordingly. Trust also helps keep innocent users out of trouble, by isolating them from newer users, where the censor's agents are more likely to be found. 3) Highly trusted users are allowed to *recommend* their friends to Salmon. The algorithm maintains a social graph of user recommendations, and assign members of the same connected component — a recommendation ancestry tree — to the same servers, whenever possible.

Other works [14,15] have employed techniques with the same overall theme of gradually identifying and punishing suspicious users. Salmon (the algorithm) is more robust than other approaches, thanks mainly to its trust levels. Our simulations show that the addition of our trust level logic significantly reduces how many users the censor can cut off from access to proxy servers. rBridge [15], the most robust previous server distribution technique, allows over three times as many users to be cut off from the system as Salmon.

Salmon (the system) has another advantage: it strikes a good balance between being easily accessible to the general public, and keeping the censor's agents out. Previous proposals have gone a bit too far in one of those directions. In addition to recommendations, the Salmon system accepts new users who can prove ownership of a well-established Facebook account. If Facebook is blocked, existing robust but low-bandwidth techniques are sufficient for getting the user through Salmon's registration process. The system is therefore open even to people who do not know any longtime Salmon users.

Salmon's name comes from its trust levels: just as salmon swimming upstream must hop up small waterfalls, and might briefly fall backwards, users need to advance up the trust levels. As a bonus, 'salmon' can be translated to Persian as 'free (as in freedom) fish'!

We assume a censor that:

- has enough employees to perform labor-intensive tasks, such as examining a list of servers.
- cannot identify proxies via traffic fingerprinting. (Anti-fingerprinting is important and has been studied [36–38], but is orthogonal to server

distribution.)

- may try to block as many servers as it can immediately, or may be willing to simply discover servers, and not take action for months.

Salmon is built to solve a single deceptively simple problem. We are in possession of some scarce resource that we wish to distribute to a large group of users, most of whom are strangers to us, some of whom represent an adversary seeking to destroy the resource. In terms of censorship circumvention, we have:

- We want to give proxy server IP addresses to any interested user living under censorship.
- The censor's employees can pose as ordinary users.
- A censor can choose to block any proxy it discovers.
- Such a block reveals to us that the censor somehow learned there was a proxy server at that address.

This problem's difficulty comes from the requirement to keep the system open to all potential users, combined with censors' ability to behave identically to good users for as long as they wish. In fact, it's clear from these two facts that a perfect solution — one that keeps all servers from being blocked — is impossible: until the censor blocks a server, its agents can remain indistinguishable from good users.

3.2 Algorithm

So long as we open our system to the general public, a perfect solution —one that prevents the censor from blocking even a single server— is impossible. Perfection is not required, however: a censor that can only block 10 out of 1,000 servers has certainly not accomplished much. Our goal is to limit the censor's ability to block servers. Blocking servers is simply the tool the censor uses to work towards its actual goal: denying our users free internet access. Therefore, we perform all evaluations in terms of how many users remain able to access one of our servers in the wake of the censor's attack. We now describe the algorithm.

Salmon's algorithm has three core components. 1) We track the probability that each user is an agent of the censor (*suspicion*) and ban likely agents. 2) We track how much *trust* each user has earned, and distribute higher-quality servers accordingly. 3) We maintain a social graph of user *recommendations*; when assigning servers, we group together members of the same connected component.

Suspicion: The censor's weapon is its ability to hide among real users while damaging the system. When three users are the suspects for a blocked server, we have no choice but to say they each are equally likely to be the culprit. That is, in a block event on a server with n users, we consider each user to be innocent with probability $\frac{n-1}{n}$. Note that we are defining "users of the server" to include every user we have given the server's address to; no matter when they last connected, a user never "leaves" a server.

Salmon's algorithm for weeding out agents builds this information into something less uncertain: the probability that a user is not an agent is the product of its probabilities of innocence from every block event it has been involved with. Throughout the paper we use the more concise term "suspicion" — the complement of the probability of innocence. We permanently ban any user whose suspicion exceeds the threshold of suspicion, T.

Trust levels: In a system like Salmon, some users can reasonably be considered more trustworthy than others: most obviously, friends and friends-of-friends of the people running the system. Additionally, a user who has known a proxy address for months without the address getting blocked seems less likely to be an agent. We base this heuristic on the censor's optimal strategy: as we will demonstrate, the censor does not benefit (in terms of maximizing users without access to Salmon) from waiting long periods of time before beginning to block the servers its agents have collected.

Salmon quantifies the concept of trust with discrete *trust levels*. All entities in the system — both users and servers — live in a single specific trust level at any given time. When the system assigns a server to a user, it will only choose a server of the user's trust level. Users can move up and down over time. Servers only move up.

Level 0 is the entry trust level. Users not recommended by a trusted user start here. There are negative levels to accommodate level 0 groups who experience a server block, extending as far down as a user could possibly fall before being banned: users are banned solely on high suspicion, not low

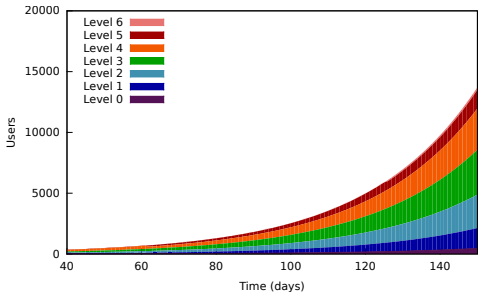

Figure 3.1: Evolution over time of the proportions of users in different trust levels. Every day, for every 30 users, one new user enters the system at level 0.

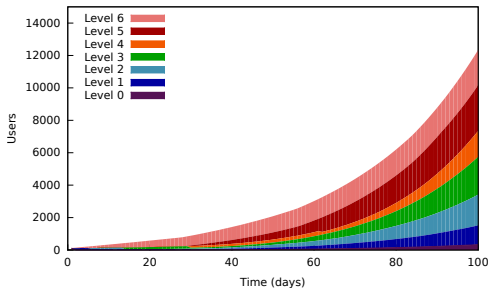

Figure 3.2: Similar to Figure 3.1, but the system launches with 20 special users who are considered to be above the trust hierarchy, and can recommend one level 6 user per day. Level 6 users recommend one level 5 user every four weeks.

trust. The levels reach up to a maximum of L, a parameter to be decided upon.

A user loses a trust level every time a server whose address it knows is blocked. A user gains a trust level if it goes for a period of time without its assigned server being blocked. That period doubles every level: promotion from level n to $n+1$ takes 2^{n+1} days, *e.g.* promotion to level 6 takes over two months. (Levels ≤ 0 all take one day). We chose the exponential durations to strike a balance between quickly lifting good users out of the less desirable bottom levels, and protecting the highest levels. As we describe in the next subsection, forcing the censor to endure hefty delays is a very effective defense, especially given that we allow users to join via recommendation. We chose maximum level $L = 6$ as a good compromise in the same tradeoff.

A server's trust level is the minimum trust level of its assigned users. For instance, if a new server is assigned to a level 2 user, the server has been chosen to start at trust level 2. If another level 2 user is assigned to the server, and then the first user attains level 3, the server remains at level 2. Then, if the second user attains level 3 before any other users are assigned to the server, the server rises to level 3. Of course, a server never loses trust levels, because its users would only lose a trust level if it were blocked!

A design with no uppermost trust level, perhaps with each level l able to recommend users to level $l-1$, would be an interesting alternate design. However, that design raises the question of what level users recommended

by the implicitly trusted users would join at.

The trust levels benefit higher-trust users in two ways. First, servers are not equal. Although we require a minimum bandwidth (100KB/s) from our volunteers, more generous volunteers might provide more. For instance, our own servers each offer about 1.5MB/s. Salmon assigns faster servers to more trusted users.

Second, users at higher trust levels are better isolated from censor attacks. Only a censor willing to leave servers unblocked for months at a time, or able to obtain a recommendation from a trusted user, will get access to servers at high trust levels. Our innocent high trust users are therefore less likely to experience server block events, and to be banned.

Recommendation graph: A user at maximum trust level L may recommend friends to Salmon; these friends join at level $L-1$. Users whom Salmon axiomatically assumes trustworthy (*i.e.*, friends of the directory server admins) are sent to a special level outside of the hierarchy, which is attainable only in this manner. They can recommend other users to level L, and are allowed to recommend after a much shorter delays than level L users: once per D_* days vs. once per D_L days. We have chosen $D_* = 1, D_L = 30$, as we describe in subsection 3.3.6.

We must be careful about allowing such recommendations in the system we have described thus far: recommendations give a patient censor the ability to exponentially grow a collection of agents at high trust levels. The censor could reach practically any agent/user ratio it desired. There are two ways to deal with the censor recommendation issue.

First, and most simply, we can delay the censor. Getting significant exponential growth from the recommendation system takes several months: the first wave of users must wait over four months to recommend the second wave; the second and all subsequent waves must wait over two months to begin recommending. We expect that our volunteer servers can change IP addresses somewhat easily — easily enough that many would be willing to do so once every several months, but certainly not as frequently as *e.g.* once a week.

In addition to recovering from a recommendation-based mass block, we can take prophylactic action. Users want to avoid being grouped with agents, so they should naturally want to be grouped with their recommend{er,ee}s: real-world friends whom they trust not to be agents. We use the following

logic: when assigning a new server to a user u, we first look for one that has already been given to a member of its connected component — always a tree, so call it $T(u)$ — in the recommendation graph. It is natural to group users in this way, as friends could easily share proxy login credentials among themselves regardless of our logic. Indeed, a design discussed later (Proximax [14]) is built entirely upon this fact. Because this sharing of servers among friends is so natural, this logic takes precedence over the trust levels: a user may be placed on a higher-level server if it includes a recommendation friend.

If there are no such servers with room left, we instead choose a server with enough free slots for all of $T(u)$. Enough slots on the chosen server are reserved to ensure that the rest of the users in the tree can join later. Let M be the maximum users allowed in a group. If $|T(u)| \geq M$, only a fresh, unassigned server will be used. Helpfully, this means that a group of M or more friends who build an agent-free recommendation tree receive servers guaranteed to be agent-free.

Just as the trust levels keep innocent, highly trusted users isolated from impatient censor agents, the recommendation graph tends to keep agents who were created by recommendation in groups with themselves. Both mechanisms share a fundamental purpose: keeping censor agents as tightly grouped together as possible. The more evenly the agents can spread throughout the system, the more servers they will discover.

It is possible that some users might wind up in the same recommendation tree as an agent, *e.g.* if the censor hands out free recommendation codes in an internet forum. Users should be careful about the source of their recommendations. This scenario does not spell guaranteed banning for the user u, however; it simply makes them more likely to be grouped with an agent, depending on what fraction of users in $T(u)$ are agents.

Summary: Users are given one server at a time. The server is shared among a group of users. If a user's server gets blocked, we *trust* the user less, and *suspect* them more. Suspicion is used to decide on bans for users. Trust is used to reward reliable users with better servers, and to keep them safe from the collateral damage of an impatient censor. Users gain trust if the servers they have been given remain unblocked for long periods of time. Highly trusted users can recommend a limited amount of friends to become highly trusted. In the undirected graph of recommendations, members of a

connected component are given the same servers.

3.3 Implementation

The previous section presented the theoretical design of Salmon: the algorithm for careful proxy distribution. This section deals with the real-world details of our implementation: the architecture of the system's software components, restricting account creation with Facebook, our choice of values for the tunable parameters present in our design, and a possible attack stemming from a practical issue not included in the threat model.

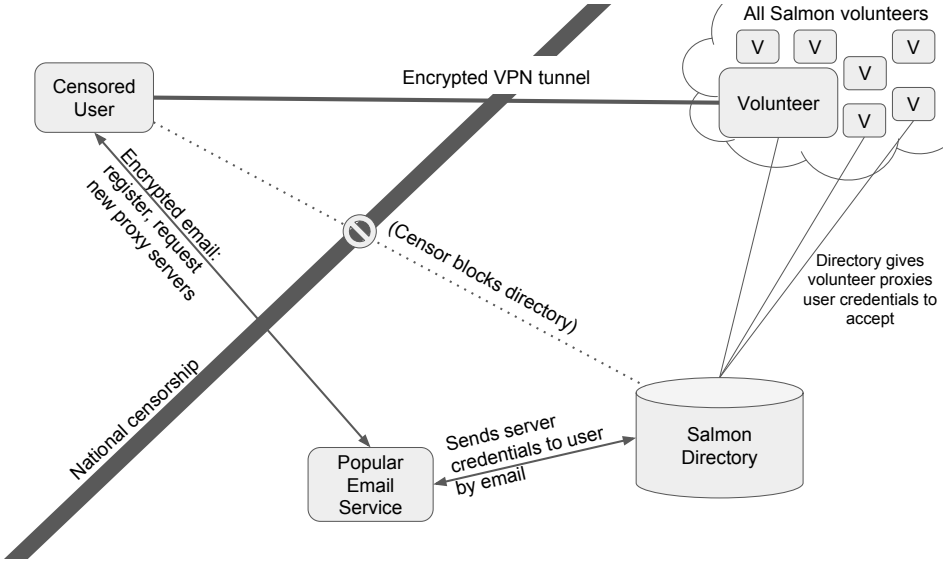

Figure 3.3: The architecture of the Salmon system. A central directory server maintains Salmon's algorithmic logic regarding proxy server distribution, instructs proxy servers to grant access to the appropriate clients, and informs clients of their proxy servers' identities. The directory server is assumed to be blocked by the censor, and so communicates with clients via any popular encrypted email service not controlled by the censor.

3.3.1 Components

Our implementation of Salmon comprises three components: a Windows client program for the users ($\sim$ 4,300 C++ SLOC, as counted by the CLOC tool, excluding GUI code); a server daemon for the volunteers (Linux and

Windows; $\sim$ 1,400 C SLOC for the Linux version); and a central directory server to keep track of servers, and distribute their IP addresses to users ($\sim$ 2,700 D and 1,000 C++ SLOC, using MongoDB for storage). All of the components have been released under the GPLv3, and are available at `https://github.com/SalmonProject`. Figure 3.3 illustrates how these components interact.

The entirety of the algorithm described in this paper is carried out by the directory server. Since the censor can easily block the directory server, the client program communicates with it via any commonly used encrypted email service that the censor does not block (or control). The email address used to communicate with the directory server serves as a Salmon account's identifier.

Underneath our own client and server programs, we run SoftEther VPN. SoftEther is mature, and comes tested for censorship circumvention by VPN Gate [12]. Throughout the paper, we discuss distributing "proxy server addresses." In reality, the directory server also provides the Salmon client with the server's X.509 certificate, and with login credentials, which it must also instruct the server in question to accept.

We want Salmon to be easily available to all censored internet users, not just technically adept ones. Our client implementation is for Windows, is compatible back to XP, and is available in English, Chinese, and Persian. The client can also export its list of VPN credentials to iOS (in convenient one-tap `.mobileconfig` files) and Android devices.

Our algorithm is vulnerable if the censor can create an arbitrary number of user identities. After such a censor begins attacking, any server that has not already received a full load of innocent users will be blocked. Therefore, we must be very careful about how we allow new users to join the system. We accept new user registrations in two ways, which we now describe.

3.3.2 Facebook Registration

A user can create a new Salmon account by demonstrating ownership of a valid Facebook account. In our initial deployment, a "valid" Facebook account is simply one that existed before 2015. Any enhancements to Facebook's screening of fake accounts would make Salmon more robust. Cat-

and-mouse games could be played with account requirements, such as post frequency or patterns. For a more sophisticated defense, the system could attempt to shut out accounts with fake profile pictures. For instance, it could require profile pictures to show a face, check via reverse image search that those pictures do not exist outside of Facebook, and reject any Facebook account whose face matches other accounts that already created a Salmon identity.

Of course, Facebook is itself a prime censorship target. Low bandwidth covert channels, such as through Skype [28] or email [29] are helpful here. While their performance is not quite what users would want for general internet usage, they are sufficient for Salmon's one-time registration process, with its single Facebook post.

To demonstrate ownership of an account, a user first sends the directory server a link to the Facebook profile they want to register with, and a post that they intend to post to their wall in the future. To hide the fact that the user is using Facebook to register for Salmon, no messages or friend requests are required to be exchanged with any special Salmon Facebook accounts. Rather, the user's account must be publicly visible during the registration process. This is an unfortunate requirement for users who prefer to keep their Facebook profile private, but the profile need only be public for a few minutes.

To prevent double registrations, the directory server must remember which Facebook profiles have created Salmon accounts. Since our logic must be able to test a Facebook profile for membership in the set of already used profiles, an intruder who gains control over the directory server will be able to do the same. To prevent the intruder from going beyond membership tests, and obtaining a full list of Salmon Facebook profiles, we store the links hashed. Note that the censor cannot test membership simply by attempting to register with the user's Facebook profile URL, unless it can guess the user's next Facebook status update.

Our use of Facebook is not guaranteed to limit the censor to one Salmon account per employee. Iran, in particular, is known to already be using realistic fake profiles to try to make connections to Iranian Facebook users, who are typically much more careful with privacy settings than the typical Facebook user. In the case of China, "one account per employee" might actually be overwhelming for a Salmon deployment with hundreds or thousands

of servers, given the size of its censorship operation. (That said, those employees would be vastly more likely to have active, well established accounts on Renren than on Facebook). In such a case, we could disable Facebook registration, so that new users could only join via recommendation. The closest design would then be rBridge, relative to which Salmon would still provide an easier and more reliable path for new users to join, even without non-recommended account creation.

Therefore, the Facebook registration mechanism can be removed without dismantling Salmon; it is an additional feature meant to provide openness equal to VPN Gate, while being significantly harder to block. Since VPN Gate is currently successfully serving users (as of May 2016), this is a good goal. A recommendation-only version of Salmon would still serve users, and until that becomes necessary, the system is open to the public.

Even if the censor suddenly adds many agents and discovers many servers before we realize we need to disable Facebook registration, we can have servers reset IP addresses once Facebook registration is disabled. In the time before we are forced to disable Facebook, we grow much more rapidly, and with a more widely seeded social graph, than we could with only recommendations.

3.3.3 Recommended Registration

A user can create an account at the recommendation of an existing user who is sufficiently trusted by the system. This process is simple: the directory server generates a short alphanumeric code for the recommender to give to the recommendee, and when the recommendee registers, they provide the code rather than proving ownership of a Facebook profile. The recommendee must not begin its life immediately able to recommend, or else a censor with a single recommender account could instantly create arbitrarily many identities. The details of when recommendation is possible, and what the result is for the recommendee, are described in §3.2.

Also described in §3.2 is the fact that part of the algorithm involves tracking who recommended whom. That is, we are storing some subset of a social graph of our users, perhaps making the machine storing it a target. However, an attacker gaining access, especially a national censor, would in fact not find

much new information. The graph stored is a tiny shadow of a true social graph. The graph is a forest of trees, with each user having at most one incoming recommendation edge. With $n - 1$ edges for n users, a recommendation forest would have fewer than 1% of the edges of the corresponding set of Facebook users.

Additionally, national censors may have other options for collecting this sort of data. Of the two countries we are most interested in, China and Iran, only Iran might need to resort to breaking into servers to obtain social network data about its citizens. Renren, the Chinese Facebook clone, is as ubiquitous in China as Facebook is in free countries. The Chinese government certainly has access to all of Renren's data; there is not much more they could want.

Finally, we are not directly storing edges between Facebook IDs, but rather between email addresses, which we suggest to be a fresh account just for Salmon, in case a censor distributes malicious copies of Salmon to steal passwords. An email address is linked to a hash of a Facebook ID (if the user registered with Facebook). While these measures do not make meaningful social connection data irretrievable, they do mean that an intruder will not directly find a subset of Facebook's graph. In particular, the hashing means that if the intruder does not already know that a certain Facebook account exists, they will not know what they are looking at when they see it in the graph.

3.3.4 Detecting Blocked Servers

In addition to being blocked by the censor, servers can also simply go offline, as many volunteer servers will be individuals' personal computers. It is crucial to distinguish between these cases, or else churn (to say nothing of a censor providing intentionally short-lived servers) would cause many users to be needlessly banned.

We want to view a server as "blocked" if it is still functioning perfectly fine, and connections to it only fail due to a censor's interference. The censor only controls traffic within its own country, and so we can conclude a server is blocked if and only if a host outside the censoring country can reach the server at the same time that the client cannot. The check is done by the

directory server.

Experience has led us to cover some corner cases, such as a server coming online just as the client reports it to be down, or a client with intermittent internet access. Unfortunately, putting this issue to rest on the theoretical side requires more than patching a few bugs. If there is an ongoing non-zero probability that a user's server will be erroneously classified as blocked, then eventually that user is going to be banned. We hope our implementation is robust enough that these erroneous bans would take years, but without formal verification of the entire system, we cannot be sure. Ideally, some more sophisticated approach, which would take these observations as input, and prevent or reduce false positives at the cost of increasing false negatives, should be used. The development of such an approach would be an interesting avenue of research.

3.3.5 Server Churn

Salmon prevents churn from causing erratic availability by adding new servers to a group — without penalizing the group's users — when all of the group's servers are offline, but not blocked. Users stay on the same server until it goes offline (and is verified by the central directory as offline not blocked), at which point they are entitled to receive a new server. They also retain access to the previous server, and can choose to switch back to it whenever it comes back online. When told to connect, the Salmon client attempts connections to its collected servers in order of a rough heuristic score of advertised bandwidth and observed RTT. Servers observed to be offline within the last week are tried after all others.

While the assignment of multiple servers to a single group adds significant complexity to the implementation, it does not change the algorithmic concepts. A group of servers with at least one online at any given time provides essentially the same service as a single server that is always online. All clients in the group are entitled to learn about all of the group's servers, and no clients outside the group are entitled to learn any. Therefore, for purposes of analysis, we assume that each group of users receives a single server, which is always usable until blocked by the censor. We refer to users being "grouped together" or "assigned to a server"; these terms are synonymous.

Additionally, some servers, such as those run on home internet connections, may periodically change IP addresses. Our implementation allows the client to stay with such a server. The server keeps the directory informed of its address, and when the client tells the directory that the server is offline, the directory simply informs the client of the new address.

This IP address churn can in fact be extremely useful to the system, if it can be intentionally induced. A blocked server can "return to life" by simply changing its IP address, and reconnecting to the directory server (which will of course not assign it to the exact same group of users that got it blocked). A volunteer can provide an email address at which to be notified if their server becomes blocked. The email includes instructions for forcing a typical home internet connection to change IP addresses.

3.3.6 System Parameters

Our implementation assigns up to 10 users to a single group, and asks servers to provide a minimum of 100KB/s. We hope that many of our volunteers will provide more than the minimum; the servers we are ourselves hosting each provide around 1.5MB/s.

Ultimately, the question of group size comes down to a fundamental trade-off: serving more users with a given number of servers on one hand, and providing more bandwidth and limiting the censor's ability to block servers on the other. We wanted to push the group size as high as possible while both staying safe from the censor, and providing users with good throughput. Given our minimum bandwidth requirement of 100KB/s for the servers, if all clients in a group of 10 attempt to transfer large amounts of data simultaneously over TCP, they will each get around 10KB/s. We did not want to design into the system the potential for users to see single digit KB/s bandwidth: dial-up, which any modern VPN-based anti-censorship technique ought to outpace, is 7KB/s. Therefore, on the implementation side, we decided on 10 as the upper bound. On the algorithmic side, our simulations using size 10 groups were sufficiently robust.

Our simulations led us to set the suspicion threshold $T = \frac{1}{3}$. For our maximum group size $M = 10$, users are banned after witnessing 4 server blocks in full groups.

We chose highest level $L = 6$ as a compromise between allowing users to become highly trusted in a reasonable timeframe, and keeping the censor recommending agents at a slow rate. Similarly, we chose $D_* = 1$ day and $D_L = 30$ days for the wait between recommendations for special users and level L users, respectively.

3.3.7 Volunteer Safety

A Salmon server's logs are an appealing target for the censor. The censor could gather Salmon logs with much less effort by running honeypot servers, but volunteers should still be aware of this point. More immediately, as a side effect to providing users with unfiltered internet access, Salmon causes users' actions to appear to come from the server they are using. This obviously has the potential to cause trouble for the server volunteers, which must be prevented.

We do not know how various legal systems would ultimately parse this issue, but logically speaking, a Tor exit node and a volunteer VPN server ought to be equivalent in terms of liability. A Tor exit node breaking laws is indistinguishable from its users breaking laws. A VPN volunteer who breaks laws and fabricates logs showing their users doing it is similarly indistinguishable. We therefore expect VPN server volunteers would be at least as safe as Tor exit node providers.

Tor has mostly managed to function without anything terrible happening to its exit node providers — outside of Austria, where an exit node provider was judged criminally responsible for the traffic processed by his node [39]. Given Tor's record, and the fact that, unlike Tor, our volunteers have logs to present if they are ever accused, we feel confident that our volunteers are safe. If a volunteer wishes not to keep logs, either because they feel safer that way, or because they want to provide more protection to their users, SoftEther's logging can be turned off. Indeed, in VPN Gate (whose volunteers' logging policies are listed), a tiny minority of volunteers do not keep logs.

BitTorrent deserves a special mention, as the most likely source of abuse. People in censored countries might not disconnect from Salmon before torrenting. Worse, people in uncensored countries would certainly use Salmon not to evade any true censorship, but just to hide their IP address while tor-

renting unauthorized material. Salmon volunteers would therefore be likely to receive a steady flow of copyright infringement complaints. Even if no volunteer was ever sued, frequently being accused and needing to dig information out of logs would be prohibitively intimidating and annoying, and their ISP would likely make them stop.

We therefore decided to block Salmon users from using BitTorrent. BitTorrent is generally unaffected by Iranian and Chinese censorship, so this is not a harsh measure. We block BitTorrent in the manner suggested for Tor exit relays [40]: by allowing the client to send packets only to a few whitelisted ports, comprising the standard ports for the internet services most used by typical users: DNS, HTTP, HTTPS, FTP, ssh, and a few common instant messaging clients. Skype is another critical service, but it is satisfied with access to the HTTP+HTTPS ports (80 and 443). BitTorrent clients typically choose a random high-numbered port to receive packets on, and peers unable to send to those ports have no way to get them to change. Being cut off from the vast majority of BitTorrent clients renders BitTorrent useless.

3.3.8 Privacy

Some circumvention approaches build on top of Tor, or otherwise attempt to provide users with privacy / anonymity. As a one-hop VPN-based approach, which furthermore relies on unvetted volunteers to provide servers, Salmon users should not feel private. We clearly convey this fact, including the example of potential government honeypots, which should get users' attention (and is easily the greatest threat). Privacy would be ideal (and the lack of privacy disqualifies Salmon for truly dangerous activities), but given the prevalence of paid VPN services as a circumvention method, Salmon is not any worse than the status quo.

3.3.9 Usability

We want Salmon to be easily available to all censored internet users, not just technically adept ones. Our initial client implementation is for Windows, and is compatible back to XP, which retains a large install base in China. A SoftEther client is available for Linux; if there is sufficient demand, the

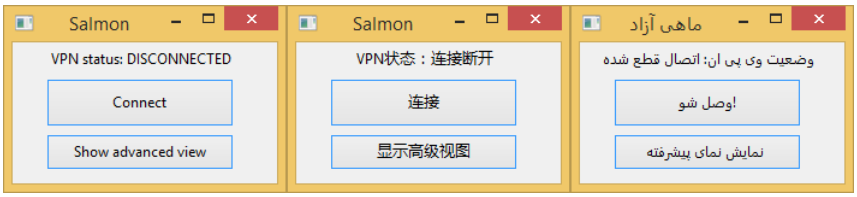

Figure 3.4: The interface of a logged in Salmon client. The advanced options include recommendations, and sending server credentials to mobile devices.

Salmon client can easily be modified to run under Wine, as its Windows-specific code is almost entirely simple GUI components.

The client runs on top of SoftEther, and takes care of all configuration tasks. Even when the client has amassed a list of servers, the user is presented with a single simple "Connect" button (figure 3.4), and the Salmon client finds the best available one to connect to. The email-based communication with the directory server can be handled automatically by the client, using the vmime library. There is also a manual email option as a fallback. The client is available in English, Chinese, and Persian.

The client can export its list of server credentials to iOS (in convenient one-tap `.mobileconfig` files) and Android devices, as SoftEther supports L2TP IPSEC.

3.3.10 The Zig-zag Attack

Proxy-based techniques without careful distribution, such as VPN Gate or the direct distribution of Tor [13] hidden bridges, face a problem known as the zig-zag attack [41]. In this attack, the censor somehow discovers a few servers, and watches their citizens' traffic for connections to those servers. Citizens communicating with the servers are assumed to be using the same circumvention system, and so any other IP addresses they communicate with are worth close inspection, since there are likely to be some new proxy servers among them.

This exact implementation of the zig-zag attack does not affect careful distribution schemes such as Salmon, because clients are not free to try any server whenever they wish. However, a variant, which we will call the *active zig-zag attack*, does apply. The difference is simple: the censor now blocks

a server once it has finished observing its users, *forcing* the users onto new servers. In addition to discovering servers, the attack would damage the reputations of the innocent users who had been placed on those servers.

Defending against the zig-zag attack entails making it difficult for the censor to confirm the presence of a proxy server. Such a defense requires both authentication and camouflage. If the server cannot verify that a given connection is coming from a user who is "supposed" to be talking to them, it has no choice but to act as a functioning proxy, fully exposing itself to a zig-zagging censor. The server's behavior after deciding to reject a connection is just as important: if it doesn't look like a legitimate, innocuous server, the censor might still block it. The situation then becomes a steganographic arms race.

Each Salmon user has unique login credentials for each of their servers. Connecting to a server without valid credentials yields a page from what appears to be an ordinary HTTPS server (since credentials are requested via HTTP authentication). Of course, the issue of camouflage goes beyond authentication behavior, most notably to traffic fingerprinting. However, counter-fingerprinting techniques apply to just about any circumvention system, including those based on Tor.

The censor could also take another, more limited approach to zig-zagging, in which the discovery of one server in a group can reveal the rest. Suppose the censor is secretly in control of a popular site that it censors, and has the site give cookies to visitors. A Salmon user (with one server known to the censor) who repeatedly visits this site via different VPN servers (due to churn), presenting the same cookie with each visit, will prove to the censor that the IP addresses seen presenting the cookie are VPN servers. Note that not all of these VPN servers, even the originally discovered one, need to be Salmon servers for this attack to be effective. To mitigate this threat, users should use incognito/private browsing while on Salmon, and should not leave a browser session open for days at a time.

3.3.11 Deployment

We have a small initial deployment of Salmon running, with a few servers and about a dozen users in Iran and China. After working through some

bugs, such as unstable internet connections causing the client program to wrongly report a server block, the system is working for these users. Traffic fingerprinting therefore does not appear to be in use.

Our most interesting experience involved the WiFi at one user's university. The WiFi is public but with very limited bandwidth, except for connections to the school's VPN. However, connecting to another VPN prevents the use of SoftEther in its default configuration, short of using a VM. There is no particular indication that the purpose of this setup was to interfere with circumvention; one of the authors attended a school in a free country with the same setup. This problem can be solved by having the client's traffic enter the tunnel at a higher layer, such as with SOCKS. The SoftEther client has a SOCKS mode, although SOCKS entails additional configuration, and should not be our default.

3.4 Evaluation

To demonstrate that Salmon can withstand an attack by a realistic censor, we simulated users and censor agents requesting proxy servers. We measured how many servers the censor can block, and how many censor agents and innocent users are banned. Ultimately, though, the censor cares most about how many users are left without access when it has done all the damage it can. This quantity (as a fraction of all users) is used for the vertical axis of the charts.

To ensure that the simulations are faithful to the real system, the simulation environment was built around our implementation of the central directory server: the component of the system where all of the algorithm logic lives. Specifically, functions that would be called upon receipt of an email from a client, or a message from a server sent through TLS, are instead called directly from the simulation framework.

The simulation iterates through one day at a time, having each user check whether they have access to a server, and requesting a new one if not. Agents behave similarly, and additionally have the opportunity to block any server they have been given, whenever their strategy dictates.

Thanks to the simplicity of the simulation framework, and the reuse of code (with the algorithm logic in particular remaining unchanged), we are

confident that even if a coding bug made our real-world implementation weaker than the conceptual design, that weakness would be reflected in the simulations.

All simulations have 10,000 users, including the censor's agents, which comprise $\{1\%, 2\%, 5\%, 10\%\}$ of all users. Each of these percentages represent a serious attack by the censor: even at the lowest level, the censor has assembled 100 real Facebook accounts, and gone through the Salmon registration process for each one. The users join the system in the pattern depicted in Figure 3.2, representing exponential word-of-mouth growth, as well as a steady flow of recommendations from the small number of "special" trust level users (the Salmon administrators' personal friends).

Each simulation starts with all servers already present in the system. We vary the number of servers available to the system, from 1,000 to 2,000. The charts' horizontal axes represent a related quantity: users per server, varying from 5 to the system's maximum of 10. We ignore values beyond 10 because even without a censor attacking, lack of servers would rapidly inflate the fraction of users cut off from servers; Salmon simply stops accepting new users. In our implementation, a user who registers when all servers are full is informed of the lack, and encouraged to ask any friends they have in free countries to consider becoming Salmon volunteers. As a fallback, they are given 16 confirmed (by the process in §2.1.1) VPN Gate servers.

Allowing newly registered users to go without Salmon servers when no empty servers are available has a major benefit for the older users. If a Salmon system can grow for a time without the censor inserting agents, an increasing number of users actually become *invincible* to (having their servers discovered by) the censor. Suppose the system gains a significant initial userbase by the time the censor takes notice and begins inserting agents. Salmon fills servers one by one, rather than evenly distributing users among all available servers, and server groups have a hard cap on membership. Therefore, any user whose server group is filled before agents begin joining is invincible to being banned. To keep our simulations both simple and pessimistic on Salmon's robustness, we assume that the fraction of users owned by the censor does not increase over time; even users joining on the first day have $\{1\%, 2\%, 5\%, 10\%\}$ of agents among them.

While the system's logic — whether to give a user a server, which server to give, when to ban a user, recommendations — is deterministic, the simula-

tions are a random process. The order in which users (with agents mixed in) act is randomized for each run of the simulation, leading to different groupings of users, and therefore different effects from the blocking of servers. We ran all of our simulations multiple times; the points on our charts are means, and the bars are 95% confidence intervals. 10 replications were enough to shrink most of the 95% confidence interval bars smaller than the charts' point icons.

3.4.1 Adversary Strategy

The most important aspect of this sort of simulation is the behavior of the adversary. Our simulated adversary should follow the strategy that best achieves the real-world adversary's goal. Against Salmon's distribution strategy, a censor can essentially never block every single citizen from using Salmon; because server groups fill up, a single full group without agents is safe forever. The adversary therefore has the quantitative goal of maximizing the fraction of users whom it can cut off from the Salmon system.

Due to the design of the system, one agent represents four opportunities to block a server (assuming the agent only blocks when its group is full). Therefore, a censor with A agents has the potential to block up to $4A$ servers. A server needs only one agent assigned to it for the censor to be able to block it. Blocking a server costs the censor one blocking opportunity for every agent who had been assigned to the server, and therefore every agent beyond the first in a given group is a waste. Then the optimal course of events for the censor is for all agents to always be alone in their groups.

Once an agent is in the system and has requested a server, the only action it can perform is to block its current server. As long as the agent's server is not blocked, nothing besides trust will change for the agent and its group. The agent will never see any other servers, and no other users will join (once the group has filled) or leave the group. The censor can only control when its agents will be in the market for a new server, and therefore has two levers to build a strategy with: when agents first join the system, and when they block servers.

If all of the agents joined at once, all but the first and last few would be placed in all-agent groups. In this situation, a quarter of the censor's blocking

opportunities have already been expended in the worst way possible: not only have a minimum of servers been discovered, but there has been no collateral damage to legitimate users. On the other hand, the censor must be sure not to insert agents too slowly relative to the rate at which real users join, or multiple groups will fill entirely with real users, becoming invincible (within our distribution system). Of course, if there are fewer agents than groups, this is inevitable, but lagging behind the rate of joining users exacerbates the problem.

In the real world, learning this rate should be difficult for the censor, as we do not publish user statistics. It is overwhelmingly unlikely that the censor would be able to land its agents exactly in every tenth spot. For evaluation purposes, the most natural compromise between this perfect censor, and one that has all agents join at once, is the only other simple configuration: a uniformly random permutation of the order in which users (including agents) join, while agents are joining.

The other question of censor strategy is the timing of server blocks. It turns out that having all agents block simultaneously does not clump them in the same way that having them all join simultaneously would. At the time of the mass block, there is a certain distribution of users and agents, based on when the agents joined, across levels and groups. A mass block event simply reshuffles the group component of that distribution.

3.4.2 Benefit of Trust Levels

To show the effect of Salmon's trust level logic, we ran two sets of simulations: one with the standard trust level logic (Figure 3.6), and one without (Figure 3.5). In both cases, we assume the censor is not patient enough to wait over four months for its agents to be able recommend more into the system. Users enter the system in the pattern described in Figure 3.2, and the censor begins attacking when the userbase (sans agents) reaches 10,000: day 94.

Because users leave level 0 after just two days, its membership is consistently quite small. The censor's agents are more effective the more evenly distributed throughout the system they are. Therefore, having all agents join at once, or initiating blocking while most agents are at the same level, would

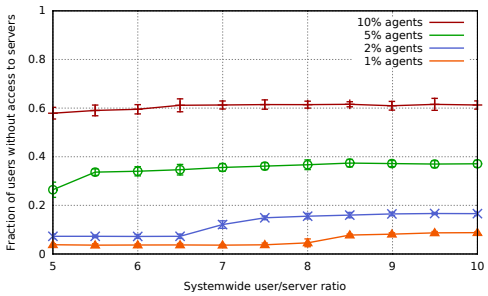

Figure 3.5: This simulation omits the trust level mechanism, benefitting the censor. Vertical axis represents users without access to servers after the censor has blocked all the servers it can.

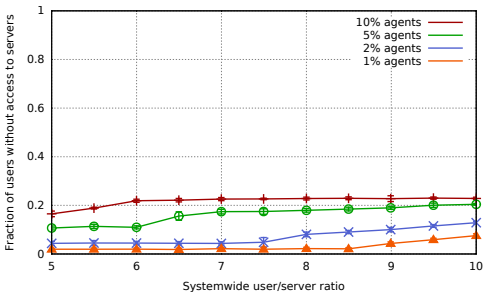

Figure 3.6: Same as Figure 3.5, but with trust level logic enabled. The users join the system in the same pattern as in Figure 3.2.

accomplish very little. We therefore assume the censor is at least somewhat patient: it is willing to wait long enough to distribute its agents into levels 0, 1, and 2. (Recall that we fight more patient adversaries with the servers' ability to occasionally change IP addresses.)

3.4.3 Recommendation Attack

In the algorithm design section (section 3.2), we discussed the possibility of a patient censor using the recommendation system to grow a huge network of agents. We have one solution in place: forcing this process to cost at least several months of inactivity on the part of the censor, and then having volunteers change their IP addresses after the mass block. However, we should also attempt to minimize the number of servers that would be affected by such a mass block in the first place. We therefore checked whether our strategy of grouping users by recommendation helps in this regard.

With our chosen delays for trust level promotion and recommendations, it would take an extremely patient censor to achieve true exponential growth. The initial batch of agents must wait over 4 months to be able to recommend, having started from level 0, and the agents they recommended must wait over 2 months to make recommendations of their own. The censor must therefore wait for well over 6 months before the growth even begins to be exponential. If the censor must wait this long, we have already won. The censor has allowed an effective censorship circumvention system to be freely available

to its citizens for over 6 months. Furthermore, even when the censor does mass-block servers in this manner, this will be a very infrequent event, so it will not be much of a burden for our server volunteers to acquire new IP addresses, as described earlier.

Therefore, we consider an only somewhat patient censor, who is only willing to wait long enough to recommend one or two additional agents per original agent (requiring a wait of 4 or 5 months respectively). Even if agents make just one recommendation, their ranks are doubled; a censor may judge the wait to be acceptable in exchange for that benefit. Figure 3.7 shows the benefit of our logic of grouping users from the same recommendation tree together: larger initial batches of agents can cut off from Salmon the majority of users, even up to over 95%, if this logic is not in place. The system's weakness when not enforcing the recommendation grouping is due to lack of assistance from the trust levels: against the somewhat patient adversary, the agents and users will all have reached the highest trust level together by the time the blocking starts.

3.4.4 Patient Censor

Of course, a censor may choose to have its agents lie dormant, appearing to be innocent users, and thereby save the ability to block many servers simultaneously for an emergency. Although we can rely on our IP address change defense for eventual recovery in such a situation, the bulk of the blocked servers would likely remain blocked for a day or two, while volunteers got around to changing their IP addresses. In a fast moving situation, the censor might be satisfied with just one or two days of disruption.

This is a hard problem, and unfortunately one that can potentially affect any circumvention system: if the censor develops some effective countermeasure, it can wait to deploy it until it will have high impact. For instance, VPN Gate is described as having evolved through a cat and mouse game with the Chinese censor. According to its statistics page, it is currently successful in serving users in China. Given how easily we were able to automatically enumerate the real VPN Gate servers (§2.1.1), it seems unlikely that the censor was permanently stumped. The censor may have decided to stop putting effort into the cat and mouse game, saving its next move for when it badly

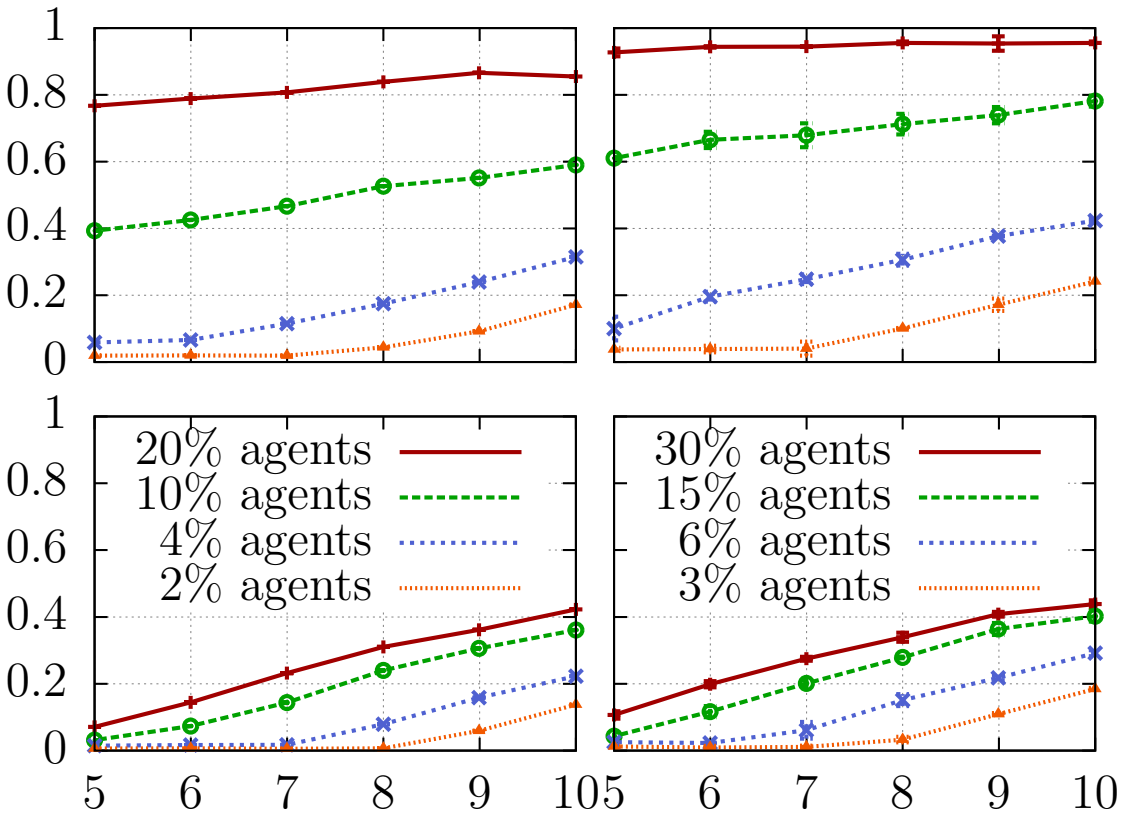

Figure 3.7: Agents and users all start at trust level 6. Agents each recommend {1,2} new agents (left, right). Users {are not, are} grouped to servers by recommendation tree (top, bottom). Agent percentages refer to final fraction after recommendations.

needed to shut down VPN Gate for a specific reason.

This is an important problem, and a solution would make anti-censorship systems considerably more robust. A good solution would have to mutate a circumvention system without outright replacing it: if the fallback is something that could run alongside the normal system, rather than in place of it, then choosing to hide it from the censor is simply a tradeoff between robustness and capacity. A system such as Salmon could make that tradeoff by keeping a reserve of unused servers.

Salmon does not fully solve this problem, but it can at least mitigate it somewhat. Users whose server groups are filled before the censor starts infiltrating agents into the system are guaranteed to stay safe, with their servers remaining unblocked. In this way, at least some core of trusted users would retain their access during the censor's emergency blocking.

3.4.5 Comparison with rBridge

We compared Salmon to rBridge, the most robust similar proxy distribution proposal. We adapted the simulations of Salmon depicted by Figure 3.6 to the conditions of rBridge's original evaluations: censor agents join by being recommended by innocent users, rather than via Facebook. In Salmon's case, this means they join at level 5, rather than 0, and have tied themselves to an unfortunate innocent user with the recommendation logic. We consider the same 5% agent case as in the original rBridge evaluations.

We changed rBridge's group size from the original choice of 40 to Salmon's choice of 10 (the group size considerations are the same for both Salmon and rBridge). We assume the censor will wait just long enough for its agents to earn enough rBridge credits to request 2 additional servers. We assume that the innocent rBridge users have accumulated infinite rBridge credits: they always have the right to request a new server.

rBridge gives all users 3 servers for robustness to churn, which Salmon accomplishes by giving free replacement servers for down (but not blocked) servers. rBridge cannot have the robustness to churn extracted from its algorithm logic: due to the requirement to keep user-server mappings secret from the directory server, users cannot report an offline-or-blocked server to the directory for it to confirm which is the case. Querying on a specific server

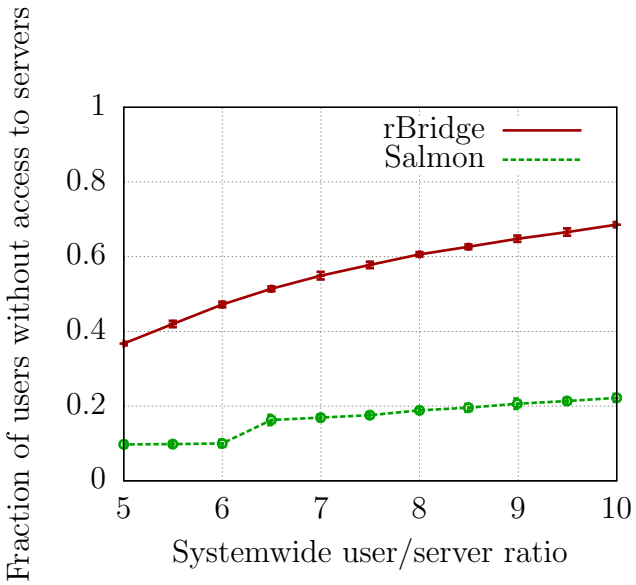

Figure 3.8: Comparison of Salmon and rBridge. The censor attacks as in the rBridge paper: by being recommended by innocent users. 5% of 10,000 users are agents.

requires anonymity, unlike receiving a random server, which can be accomplished with non-anonymous OT. If the user's only Tor bridge is unavailable, it cannot have an anonymous conversation with the directory. Therefore, rBridge must treat server churn the same as the censor's blocking, incurring the damage to users of a seemingly "blocked" server when Salmon would incur none. We therefore evaluate rBridge with its suggested parameter of 3 servers per user.

Figure 3.8 shows the results of our comparison. Salmon is much more able to weather the censor's attack than rBridge, losing fewer than a third as many users as rBridge: 22% vs 69% at the highest system load. Salmon's trust levels and recommendation grouping keep the agents corralled among a small minority of the users. This limits the damage they can do, especially at user/server ratio 10, where *no* blocked server can be replaced, and being grouped with an agent guarantees that a user will be shut out of the system. rBridge's agents are spread evenly throughout the system, maximizing the innocent users' exposure to them.

3.4.6 Steady State

The previous simulations all consider a censor who over time amasses a large number of agents in the system, and then tries to shut it down in one fell

swoop. Also of interest is the case of a censor who is content to be a nuisance, using a constant trickle of agents to block a constant trickle of servers (and perhaps with some collateral damage).

Consider the end result of the previous analyses: the censor is out of agents. Therefore, any user who has not been banned, and is in a full group, is invulnerable. In fact, they become invulnerable as soon as they join their agent-free group, even before the last agent has been banned from the system. This indicates that these simulations might give useful information even without looking at the end results with an assumption that the censor will never return. However, the important piece of information is that this fraction of users remaining is the fraction of users who survived this attack wave, and therefore all subsequent waves.

We will define an attack wave to be the agents and users who join the system during any contiguous period of time. For those who are only ever grouped with other members of their wave, the environment is identical to the previous "static" simulations, *i.e.*, the probability of being banned is equivalent to the fraction of banned users that the simulations compute. Most groups can be packed full, unless recommendation component sizes are nearly all between six and nine. Therefore, the fraction of users banned in static simulations ought to approximate the rate at which users are banned, given the current probability that new users are agents, and (relative to the rate of new users) the current rate at which servers are joining. However, we leave a fully rigorous analysis for future work. Additionally, we have not formally proven that the censor strategy we evaluate against is optimal; a formal proof of the censor's strategy's optimality would also make for a good component of a future analysis of Salmon's algorithm.

3.5 Conclusion

The key point to keep in mind about Salmon is that it is meant to provide maximum deployability. Later chapters in this dissertation describe decoy routing techniques, which are undoubtedly more powerful than proxy-based approaches such as Salmon. However, Salmon can provide free internet access to a number of censored users directly proportional to the number of free internet users who are willing to help. On the other hand, until very recently

there was no evidence that decoy routing could ever get beyond a lab-scale deployment, due to ISPs' reluctance to get involved with something that could potentially greatly damage their business.

CHAPTER 4

GHOSTPOST: DISTRIBUTED SOCIAL MEDIA UN-DELETION

The previous chapter described Salmon, an effort to create a deployable but algorithmically robust anti-censorship system. More specifically, Salmon is meant to solve the *access* censorship problem, in which a censoring government blocks sites and services that are not under its direct control. With social media having become one of the most "important" functions of the internet, a new class of censorship problem has arisen: a censor controlling what people can say on a website that it controls. This new danger of *publication* censorship is what GhostPost, the work described in this chapter, is meant to combat.

4.1 Background

The foremost social media publication censor is China. Conventional wisdom has come to view social media as particularly well suited to facilitating social unrest. Accordingly, the Chinese government is very careful in its control of social media sites, such as Sina Weibo. A study [31] of how long controversial posts survive before deletion estimated a lower bound of 4,200 full-time human employees monitoring Sina Weibo (`weibo.com`). Whatever the exact number, the Chinese censor clearly devotes a significant amount of resources to policing social media.

Although this army of censors keeps a lid on objectionable content in the long run, each undesirable post does manage to be visible for a short time. At a traditional newspaper, a handful of known employees write articles that can be carefully monitored, perhaps even requiring explicit approval from the censor to publish. With social media, a significant fraction of the country's entire population is speaking its mind, spontaneously and constantly. Punishing everyone who says anything in any way "wrong" is infeasible, and

requiring posts to be pre-approved by the censor would be practically equivalent to shutting the platform down. Censored social media platforms will therefore always allow dissent to be (temporarily) expressed. Over 90% of the posts on `weibo.com` that are fated to be deleted survive less than 24 hours, but the median deletion time is over two hours [31]. These short periods of availability turn out to be sufficient for thwarting censorship.

GhostPost is a distributed system that relies on a community of social media users to fight censorship of social media posts. These users record (passively, via a browser extension) all posts they see, and report those they see deleted to the rest of the GhostPost system. Deleted posts are automatically inserted back into the site's HTML during users' ordinary browsing. The particular implementation of GhostPost is built for `weibo.com`.

GhostPost differs from similar projects [32, 34] in its use of distributed monitoring. In addition to the ability to scale to more accounts, by monitoring the accounts that its users follow, GhostPost's attention is focused exactly where it should be. A centralized monitoring approach must accept limitations; *e.g.*, Weiboscope [32] only monitors accounts with at least 1,000 followers. Fundamentally, GhostPost's differences from these projects are the result of differing goals. Its purpose is to restore deleted posts back into the feeds of users who were supposed to see them, which requires distributed monitoring (and selection of accounts to be monitored). The previous systems are intended for archival of censored posts and analysis of the censor's behavior, for which a centralized approach is sufficient.

4.2 Threat Model

We assume the censor is a branch of a large national government. Some level of day-to-day censorship might be carried out by the company that operates the platform, to avoid the government taking more direct control.

We assume that for a sufficiently popular GhostPost deployment, the censor will attempt to block its operation, and/or dissuade citizens from using it. In particular, the censor will attempt to identify and punish GhostPost users. We assume the censor can see every post posted on Weibo, and, as will become important in evaluating GhostPost's safety, can track which Weibo users ever *observed* a given post. The censor can join GhostPost as a user,

and see which posts it resurrects.

4.3 Design

GhostPost's purpose is to resurrect social media posts deleted by a censor. GhostPost therefore needs to be able to carry out two fundamental tasks: 1) *observing posts being deleted*, and 2) *disseminating stored resurrected posts*. If consideration is not given to how deleted posts will be collected and disseminated, users could claim to have seen any account post any message. Therefore, 3) *users must be able to verify the source of a resurrected post*. Finally, these goals must be balanced against safety: a censor might seek to punish GhostPost users, in order to deter use of the system. If all resurrected posts are visible to all users, a censor could obtain information indicating that some Weibo users were more probably GhostPost users than others. Therefore, 4) *users must be able to control who receives the posts they resurrect*.

1) Observe posts being deleted: Sina Weibo is a huge social media platform, with hundreds of millions of users posting on the order of 70,000 posts per minute [31]. Any of these posts could be deleted at any time. We do not want to miss any deleted posts, but Weibo's scale makes it inevitable. A more reasonable goal is to preserve just the posts that GhostPost users are likely to be interested in, as indicated by the lists of Weibo accounts that our GhostPost users follow. Having GhostPost users themselves watch for deleted posts is then a natural choice.

2) Disseminate resurrected posts: After resurrecting a deleted post, the GhostPost system must be able to distribute it to users. All users learn of resurrected posts from the central server. Clients' interactions with the server take place via RESTful HTTPS. These queries do not require anything approaching high performance: with only metadata and handfuls of censored text transferred, they are not bandwidth intensive, and because the client queries do not block the weibo.com page load, they are not latency sensitive. Therefore, practically any means of communication, including low-performance covert channels, are adequate.

3) Verify resurrected posts: Resurrected posts are claims that someone posted something that is no longer visible. Users can hopefully apply common

sense to fight misinformation, but the best way to trust a resurrected post is to know that it comes from a trusted friend. The central server knows which users reported which resurrected posts, and can provide that information to a retrieving user (provided the sources are willing to let that user know they use GhostPost). Adding a web of trust approach, to expand post verification beyond immediate friends, might be a good direction for future work.

To guard against a malicious central server, users sign posts they resurrect. The GhostPost extension generates a key pair. Users can retrieve their friends' public keys from the server for convenience, but are encouraged to verify them separately.

4) Limit recipients of resurrected posts: As we discuss in subsection 4.6.4, a censor tracking which accounts viewed which posts, and which posts GhostPost resurrected, can gain information about how likely various Weibo users are to be GhostPost users. A user A who is worried about this possibility can protect themselves by choosing to allow posts they resurrect to be viewed only by users they trust. If A makes this choice, a user B that A does not trust will not receive a post resurrected by A until some other user—one who either trusts B, or has not enacted this restriction—has also resurrected that post.

4.4 Centralization

Although GhostPost's most important function (monitoring for deleted posts) is performed by its users, those posts are then gathered at and distributed from a central server. Ideally, a system like GhostPost would also have storage and retrieval be fully distributed, to avoid effectively just shifting the power to censor to a new central entity. However, network-effect considerations tip the scales in favor of a centralized design, for all but the largest deployments.

GhostPost's security requirements pose unique obstacles to a peer-to-peer design. Most peer-to-peer systems, even those concerned with security and privacy, expect any user to be able to directly communicate with any other user. GhostPost users are concerned first and foremost with not being identified as being GhostPost users. For this need, even something like the Octopus anonymous DHT [42] would be insufficient: it only prevents users' identities

from being associated with the queries they make, not their very participation in the system. A peer-to-peer version of GhostPost must restrict direct communication to be among real-world friends only.

Such a social network topology, like the darknet mode of Freenet [43], would add overhead, but more importantly would result in a fragmented network in all but the largest of deployments. Good coverage relies on a large set of monitoring users, as we show in our evaluation. This fact introduces a network effect to GhostPost's utility, and artificially limiting how much of the network a user can utilize would not help.

4.5 Implementation

GhostPost is implemented on the client side as a Google Chrome extension, and on the server side as a straightforward REST service. The implementation must not reveal to an observer (such as weibo.com, or the user's ISP) that the user runs GhostPost.

4.5.1 Usability

Because the Chrome Web Store is blocked in China, we must distribute the extension ourselves. This means that rather than clicking an "install this extension" link, users must instead drag and drop the extension package from their file system onto Chrome's extension management page. Our download page includes detailed instructions about this process.

Once the extension has been installed, GhostPost requires the user to register in the system by proving ownership of their Weibo account. They are prompted to do so by a pop-up message box the first time they visit weibo.com with GhostPost installed. The proof-of-ownership process is simple: the user must correctly predict the account's next post.

After registration, GhostPost functions without any user interaction. Other than an icon in the extensions section of the Chrome browser, the user's browser is completely unchanged, including while visiting weibo.com—unless GhostPost has a resurrected post that belongs on the page that the user is currently viewing. Resurrected posts are printed in red, and include the time when they were detected deleted, as well as whether they have been verified

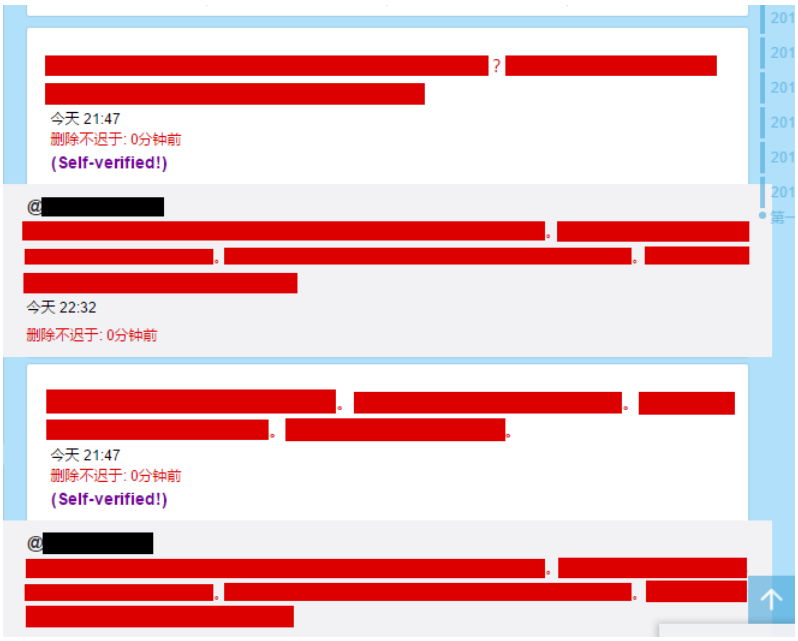

Figure 4.1: GhostPost restoring posts in-line into `weibo.com`. Usernames and post contents redacted.

by someone the user trusts. Figure 4.1 shows an example of the GhostPost Chrome extension restoring a deleted post inline in a user's Weibo feed.

With Weibo's 140 Unicode character limit, a user whose followed accounts posted 1,000 posts per day would require a (loose) upper bound of 273KiB per day, or 97.5MiB per year, to save them all. Ideally we should save posts forever, but users could forget posts older than a few weeks without doing much damage: recall that over 90% of posts are deleted within 24 hours.

Saving every image encountered on Weibo may be too much of a storage burden for many users; to be safe, our initial implementation does not store images. However, while ugly, extreme JPEG compression can reduce typical photographs to 10-20KiB, while keeping important features basically discernible. 75% of deleted posts have pictures attached (or are re-blogs of posts with pictures) [31]; the censor clearly cares about the content of images. Images might therefore be a good future addition to the system.

4.5.2 Implementation Safety

The use of the GhostPost extension must be completely invisible to the censor, both in terms of its behavior when interacting with weibo.com, and in

terms of internet activity. Google Chrome extensions are built to be completely isolated from the web pages the browser loads, unless the extension explicitly designs in hooks for the page to attach to. Therefore, GhostPost's presence is invisible to weibo.com. On the other hand, communication with the central server would be easily detectable. However, beyond detectability, we assume the central server will be blocked, and so GhostPost users must employ some form of traditional censorship circumvention to begin with.

Because the censor has control over the content of weibo.com, it could complete this registration process on behalf of any user. If we returned an error when someone tried to register an already registered account, the censor could easily test whether a given Weibo user was a GhostPost user. Therefore, we allow multiple registrations; each new registration simply adds to the list of credentials that should be accepted for that account. The fact that the censor can "take control" of someone's account in this manner is unfortunate, but not particularly damaging: all the censor could do would be to issue resurrected posts under the user's name, but with a different key from the user's real key, which the users' friends would detect.

4.6 Evaluation

GhostPost has two quantifiable aspects, which are somewhat in conflict: coverage and safety. We want to minimize the fraction of deleted posts that we fail to resurrect. At the same time, we must prevent the censor from using post resurrection patterns to find GhostPost users. We now analyze GhostPost's performance in these two areas.

4.6.1 Coverage Metric

To evaluate of GhostPost's ability to catch deleted posts, we must decide on the quantity to be measured. We only care about posts authored by users followed by GhostPost users. For coverage analysis, we lose no information by pretending that Weibo consists only of the GhostPost user base, plus the Weibo users they follow.

Having identified the users and posts that we are interested in, the natural quantity to study would seem to be the quantity of posts deleted by the

censor before GhostPost could witness them. However, this measure is not always particularly meaningful. Imagine a set of 1,000 deleted posts, among which 990 were posted by a user with a single follower, and 10 were posted by a user with 10,000 followers. If GhostPost managed to resurrect all of the 990 and none of the 10, this measure would tell us that GhostPost had 99% coverage. We must find a more meaningful measure.

Rather than considering all posts equally valuable, we will consider a post posted by a user with n followers to be worth n *postviews*. One postview represents a single user getting the opportunity to read a single deleted post that they are interested in. By this metric, we would say that in the previous example, 990 out of a possible 100,990 postviews were preserved, or about 0.98%.

4.6.2 Coverage Simulation

We evaluated GhostPost's coverage using an event-based simulation framework [44] with a scale-free topology of 1,000,000 users. The events are 1) a user posts a new post, 2) a GhostPost user saves all visible posts from a random followed account, and 3) censor deletes a post. The simulation ends when all posts have been deleted by the censor[1]. We measure the fraction of postviews restored out of the total postviews in the system.

Three factors affect GhostPost's coverage: the time it takes the censor to delete a post, the fraction of Weibo users who use GhostPost, and the interval at which GhostPost users save an account's current posts. Our implementation only has control over the last of those three. We have fixed this frequency at an average of 5 per hour, as a compromise between frequent coverage, and avoiding making the user's activity look suspicious.

A study [31] of the time it takes for Weibo posts to be deleted found that post deletion times are not very evenly distributed. First, the distribution of deletion times is long tailed; while they observed the majority of posts being deleted within 24 hours, a non-negligible fraction lived for days or weeks. More interestingly, grouping post lifetimes by the time of day they were posted at (in 24 1-hour bins) yields two clusters. Namely, fated-for-deletion

[1]Posts that are never deleted do not need to be restored, and so are irrelevant to GhostPost's degree of coverage. For purposes of evaluating coverage, we can focus only on posts that are fated to be deleted.

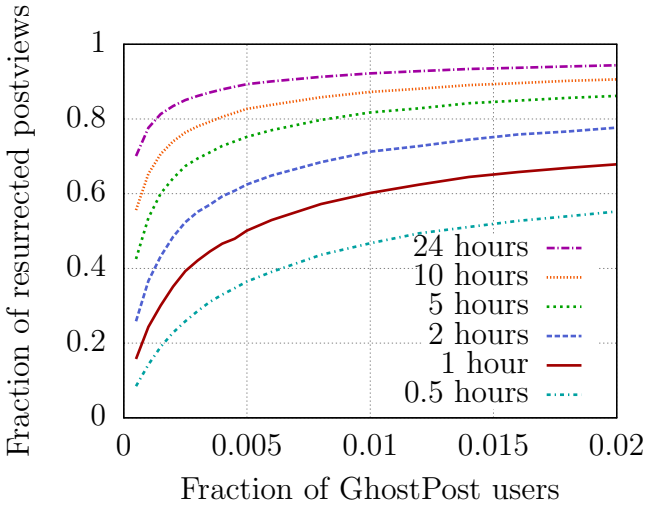

Figure 4.2: GhostPost's coverage against censors who manage to delete posts with various average speeds.

posts that are posted during the hours between 3 and 9 A.M. have median lifetimes of 8 or 9 hours, while posts posted from 10 A.M. until midnight have medians around 2 hours. The study's authors hypothesize that the censor's post-examining workforce is greatly reduced late at night, causing a backlog to build up, which is not cleared until late morning.

Because identifiable classes of posts are being monitored by censors of different capacity, we evaluated GhostPost against different censors: posts have an expected lifetime of {0.5, 1, 2, 5, 10, 24} hours. The 2 and 10 hour censors roughly correspond to the median performance of the real world daytime and nighttime censors. The 24 hour censor is an upper bound for over 90% of real world posts [31].

Because the censor is a group of employees processing a queue of posts, we model deletion time with the exponential distribution. The exponential distribution lacks real post deletion times' heavy tail, but this makes the exponential distribution a conservative choice: our results indicate that even the tiniest GhostPost deployment is extremely likely to resurrect posts that live multiple days.

When simulating GhostPost, we varied the fraction of GhostPost users in the system from 0.05% to 2%. The results of these simulations are plotted in figure 4.2. Encouragingly, GhostPost reaches nearly its full potential very quickly; steep gains are made until about 0.5% of Weibo users are using GhostPost.

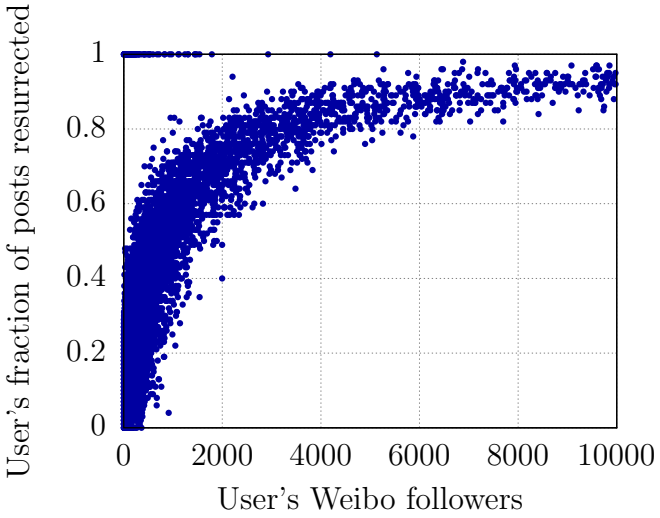

Figure 4.3: Each point is a user. The censor deletes posts after an average of 10 hours, and 1% of Weibo users are GhostPost users.

Whether a post can be resurrected is ultimately a race between the censor's agents and the poster's GhostPost followers: whichever side examines the post first wins. For a given author, GhostPost can run this race faster by either having more of its users following that author, or by having its users checking current Weibo posts more frequently. The censor can reach posts faster by hiring more employees to examine posts. However, even a highly aggressive and capable censor, one that deletes posts on average within 30 minutes after they are posted, fails to hide the majority of postviews from a GhostPost system with around 1.5% Weibo users. Sina Weibo's nighttime censor would allow a 1% GhostPost deployment to resurrect nearly 90% of postviews; the daytime censorship would allow over 70%.

4.6.3 Per-user Coverage

The previous analysis shows how likely a deleted post of interest to a GhostPost user is to be saved. In addition to that evaluation of GhostPost's ability to serve post *consumers*, we should also consider how well GhostPost serves post *producers*. A user with more followers is more likely to have more GhostPost followers, providing better coverage. We therefore examined the distribution of preserved posts for users with varying follower counts, shown in figure 4.3.

With 1% GhostPost users and a 10-hour censor, nearly 90% of all postviews

are resurrected (see figure 4.2). Most of those resurrected postviews come from users with huge amounts of followers. Users with typical numbers of followers (low hundreds) have over half of their censored posts missed by GhostPost.

However, this imbalance only applies to authors who do not care about it. Any Weibo user who wants to ensure that their posts will be preserved in GhostPost can simply become a GhostPost user, and preserve their own posts. This option is reflected in figure 4.3: points with resurrected fraction 1 are GhostPost users.

Users whose posts are deleted unusually quickly are also less covered. Users who have already had many posts deleted have lower median post lifespans than users who have only had a few deleted. The median post lifetime was about twice as long for users with fewer than 10 deleted posts than for users with 100 or more [31]. In this case as well, users who want their posts to reach the GhostPost audience should become GhostPost users.

4.6.4 Safety

We must hide GhostPost users' identities from the censor. GhostPost is not directly visible to weibo.com or the user's ISP, but observing which posts are (not) resurrected might yield information about who uses GhostPost. For this analysis, we assume that all GhostPost users allow all other users to see the posts they resurrect. The following vulnerability is eliminated if GhostPost users restrict their resurrected posts to their friends.

Imagine a Weibo account A with only two followers, F_1 and F_2, both of whom are also GhostPost users. F_1 is a legitimate user, but F_2 is an agent of the censor. If the censor deletes one of A's posts, and then F_2 receives a resurrected copy of that post, it can be sure that F_1 is a GhostPost user. Realistically, the censor will not immediately pinpoint a GhostPost user with a single post in this manner; real, active Weibo users have more than one follower.

The censor must monitor two classes of information: whether a deleted post was resurrected, and which users saw a post before its deletion. Let $\mathcal{D}$ be the set of all posts that the censor has deleted, and $\mathcal{D}_R \subset \mathcal{D}$ be the subset of resurrected posts. For each $d \in \mathcal{D}$, we will define $O(d)$ to be the

set of users who observed d, *i.e.*, all those users who loaded a Weibo page including d before it was deleted. Although the censor is only interested in $O(d)$ for $d \in \mathcal{D}$, it must track $O(p)$ for the entire lifetime of all posts: it does not know whether $p \in \mathcal{D}$ until it has decided to delete p.

To maintain $\mathcal{D}_R$, for every post $d \in \mathcal{D}$ the censor deletes, it must determine whether GhostPost can resurrect d. This requires the censor to monitor GhostPost from within: the censor must make a Weibo account, follow all accounts whose posts will be deleted, and register with GhostPost. Although a single GhostPost user claiming to follow every account would look suspicious, the censor could easily spread across multiple accounts.

Based its on observations ($\mathcal{D}$ and $\mathcal{D}_R$), the censor tracks the likelihood that each Weibo user is a GhostPost user: $\Pr[GP_u]$. The censor does not know which users are GhostPost users: that is what it is trying to learn. Therefore, when a post r is resurrected, the censor must assume that each user $u \in O(r)$ is equally likely to be responsible for resurrecting the post. Then for every new post r to be added to $\mathcal{D}_R$, for each user $u \in O(r)$

$$\Pr\left[GP_u \mid \mathcal{D}_R \bigcup \{r\}\right] = 1 - \frac{|O(r)| - 1}{|O(r)|}(1 - \Pr[GP_u \mid \mathcal{D}_R])$$

Additionally, for every deleted post $d \in \mathcal{D} \setminus \mathcal{D}_R$, *i.e.* not resurrected, every user $u \in O(d)$ has

$$\Pr\left[GP_u \mid \mathcal{D} \bigcup \{d\}\right] = 0$$

As posts are deleted and not resurrected, the censor will establish with certainty that more and more users are not GhostPost users (at the times of the deletions, at least). A non-GhostPost user remains under suspicion as long as every deleted post they observe is also observed by a GhostPost user. In other words, if a user has nothing to offer GhostPost, it is impossible to conclude from resurrected posts that they do not use GhostPost. Because almost all of GhostPost's coverage comes from the first 1 or 2% of its users, more widespread GhostPost use would allow the censor to rule out fewer non-GhostPost users. Furthermore, if some GhostPost users (perhaps those with a history of attention from the Weibo censor) are restricting their resurrections to trusted friends only, then "ruled out" users will include false negatives.

4.6.5 Post Alteration

Despite the previous point, a censor could still rule out many users in a smaller GhostPost deployment. Another countermeasure is possible. The censor must track the users who see each Weibo post ($O(p)$), and then observe whether GhostPost resurrects p once it has been deleted ($\mathcal{D}_R$). However, if the censor cannot reliably track $\mathcal{D}_R$, then it cannot be sure that $O(p)$ are not GhostPost users for $p \notin \mathcal{D}_R$.

The challenge, then, is to distribute to our users some version of the post that the censor cannot easily recognize. Weibo users evade keyword bans by using homophones, puns, characters whose components include the intended characters, and similar linguistic tricks. These tricks interfere with automated processes, while preserving the original meaning for an intelligent human reader. In this way, a user concerned about being identified as having observed a resurrected post can more safely share their resurrected posts with users outside of their friends, benefitting the system's coverage. We should also add a random offset to the post's time, and remove the author. We could prompt our users to make these modifications. There is also an automated technique [35] for generating usable Chinese homophone substitutions.

Although the homophone substitution technique would interfere with automated processes, we should also consider a censor that is investigating whether a few specific Weibo users are GhostPost users. When targeting user A, who follows some other user B, such a censor could have weibo.com display (only to A) a fake post "written by" B. Then, shortly after the next time A loads B's wall, the censor could query the GhostPost system for any new resurrected posts authored by B. A is a GhostPost user if and only if the fake post is included in the response. If a human is manually carrying out this attack, then the post alteration technique would not help much. As with other attacks, sharing resurrected posts with trusted friends only would keep the user safe.

4.7 Summary

GhostPost is designed to address an aspect of censorship that receives huge amounts of human attention from censors: the control of speech within individual social media platforms. It operates by relying on users to preserve posts from accounts that are important to them, *i.e.* those that they follow. GhostPost has been implemented and distributed, targeting the heavily censored `weibo.com`. The GhostPost design takes into account the potential for people to falsely attribute fake posts to others, by giving users a way to verify that the copy of a deleted post they are seeing was provided by someone they trust not to lie. GhostPost functions even with only one user, and quickly ramps up to defeat the majority of the censor's efforts by the time even a fraction of a percent of users have joined the GhostPost system.

CHAPTER 5

DEPLOYABLE PASSIVE DECOY ROUTING

This project sought to construct and deploy a production-quality implementation of the TapDance [10] decoy routing design. There are three issues facing prospective decoy routing deployments: 1) interference with normal traffic would essentially guarantee that ISPs would not host a decoy router, 2) ISPs are likely to become nervous about hosting a system that seems flaky and unreliable, since those attributes are smoke to security vulnerabilities' fire, and 3) ISPs handle very heavy amounts of traffic; the decoy router must be able to keep up. TapDance's design itself solves the first problem, by working with cloned packets from a tap of a link, rather than interjecting itself on the link. This project aims to solve the second problem by building a system that is demonstrably reliable, and furthermore makes extensive use of a programming language (Rust) with strong safety guarantees. Finally, we approached our implementation with the requirement that a 1U server running TapDance could keep up with a 10 Gbps link. In fact, we have found that a 1U server running our implementation can keep up with 40 Gbps.

This was a large project, including people from U. Colorado, U. Michigan, BBN, the regional ISP Merit, and the extremely popular anti-censorship system Psiphon.

5.1 Background: TapDance

A decoy router must impersonate the decoy server without causing an observer to notice anything strange about the totality of packets claiming to be from the decoy server. TapDance's passive tap faces the additional challenge of the decoy server's own response packets reaching the client, as well as the possibility that the decoy server will react strangely to the client's reactions to data it never sent.

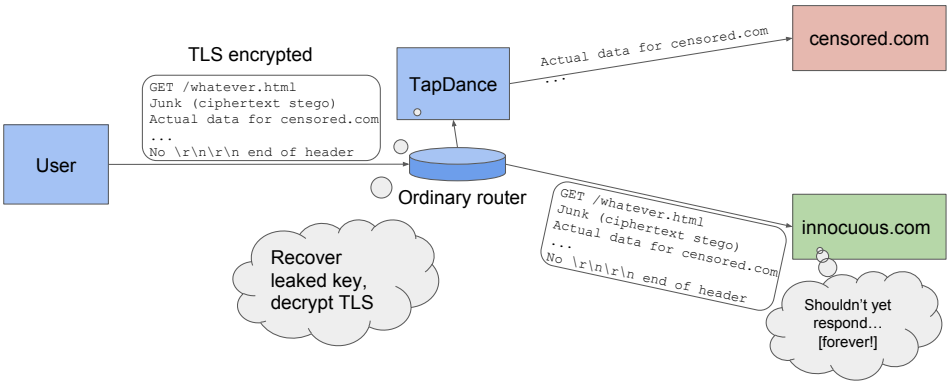

Figure 5.1: An overview of the process by which the TapDance client notifies the station of a session, and allows the station to spoof the decoy (by muting the decoy).

TapDance solves these problems by "muting" the server's HTTP logic with an unfinished HTTP header field. The decoy server will not respond until it has received a complete HTTP request, or the header has grown too long (which in practice limits TapDance to 16KB of upload per overt flow). The client can then write messages to the station into the HTTP header. However, data in the header is hidden inside a TLS tunnel that the TapDance station does not have a key for.

Therefore, the client first leaks the key to the station. It embeds in the TLS ciphertext bits the (AES-encrypted) key material for the TLS session, along with an elliptic key exchange completion that will allow the station to decrypt that key material. The elliptic curve point is encoded with Elligator [45], a scheme for encoding elliptic curve points as bit strings indistinguishable from random.

The previous two paragraphs are visually summarized in Figure 5.1.

To detect these leaked TLS session keys (called "tags"), the TapDance station must inspect the first TLS "application data"-type record of every HTTPS flow for a valid elliptic key exchange completion encoded by Elligator.

When the station starts receiving an HTTP response from the covert destination, it encrypts the data in the same TLS tunnel shared by the client and

decoy server, and returns it to the client, spoofed as if from the decoy server. Fortunately, decades ago, the architects of TCP made TapDance possible by deciding that a TCP receiver should simply ignore ACKs for data it has not yet sent. The decoy server therefore remains silent for the remainder of the TapDance session.

5.2 Implementation

To keep up with a 10 Gbps (and then 40 Gbps) link, we started out using the Intel Data Plane Development Kit. DPDK allows a programmer to (relatively) easily build what is effectively a custom driver for the network card: packets processed by DPDK never reach the kernel. However, we are doing our development and testing within a real ISP's network. This network allows jumbo frames, and furthermore uses VLANs, with the VLAN tags pushing many otherwise standard sized frames over 1500 bytes. DPDK does not support receiving these larger frames. Another framework, PF_RING, does, so we switched to it. PF_RING is a somewhat higher-level alternative with a much simpler API. Its performance is about as good as that of DPDK [46] (we used PF_RING's zero-copy mode, for maximum CPU efficiency). Ultimately, the attempt to use DPDK was an unfortunate mistake.

The other goal besides performance, reliability and security, is addressed by our choice to use the Rust programming language wherever we can manage. The Rust language has an extremely restrictive specification, designed around the concept of ownership of memory. The restrictions allow its compiler to guarantee that buffer overflows, dangling pointers, and data races will never happen. Furthermore, Rust's status as a mature but freshly designed language means that clean implementations of modern programming concepts (*e.g.* optional values and closures) are available.

The program begins in C, to set up PF_RING, but quickly switches to Rust. Figure 5.2 illustrates the TapDance station's structure, by following the journeys of a few different important types of packets. After a chunk of initialization in C, an infinite loop in C receives packets from the PF_RING framework, and delivers them immediately into Rust. This infinite loop is also responsible for calling iterations of the event loop (written in Rust) that we use for forwarding user traffic, and for calling periodic logging and cleanup

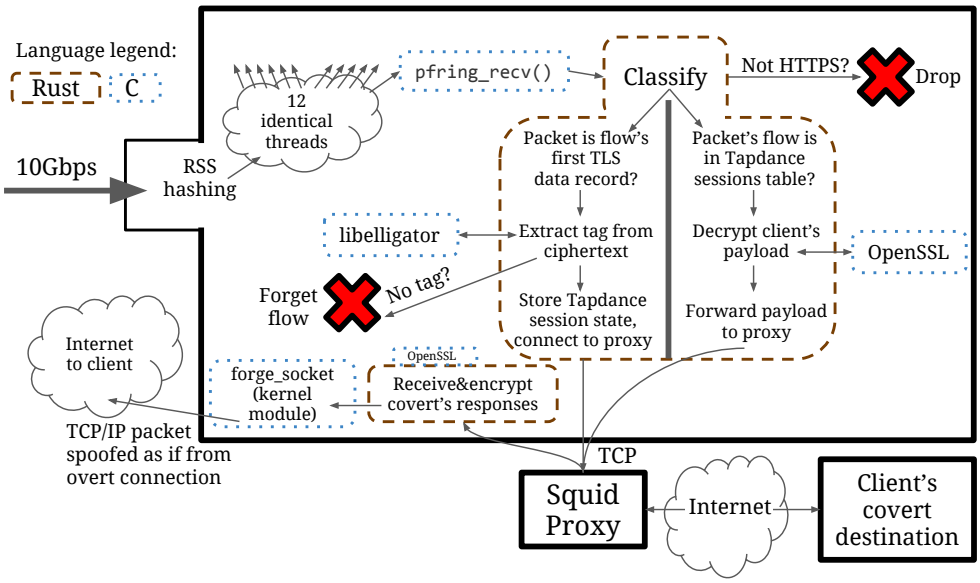

Figure 5.2: The architecture of the 10Gbps PF_RING TapDance station, showing which parts are written in Rust.

tasks (also written in Rust).

All flow state tracking logic is done in Rust. These are the sorts of tasks where Rust's memory safety is most valuable: mundane data structure allocations, insertions, searches, and deletions; things that are individually simple, but when combined grow complex enough that bugs are essentially guaranteed. By keeping this complex logic in Rust, we largely remove the possibility of a bug becoming a memory-safety-related security vulnerability.

A few parts of the program do dip back into C. These are our use of Elligator [45], an elliptic point encoding library; OpenSSL; and forge_socket, a Linux kernel module. Crucially, these calls are program-flow "dead ends": the Rust logic is simply outsourcing a certain computation to them; control returns directly to the Rust caller, rather than the C components continuing the program flow into other parts of the station logic.

Elligator is purely math with large integers. The memory safety risk of stateless computational code is low, and Rust does not yet have a native finely-tuned big-integer library, making the C implementation an easy choice.

Similarly, there is not a mature, well-tested TLS library written natively in Rust, although here memory safety is a more concerning issue. All we can do is limit the C-allocated state (to a single `SSL` object per session). Additionally, as TLS is a complex protocol, straying from a popular imple-

mentation could identify TapDance traffic to the censor via idiosyncrasies, making OpenSSL attractive regardless of reliability. We use OpenSSL to recreate the ongoing TLS session from the client's leaked secrets, and then to perform the encryptions and decryptions that constitute the rest of that TLS conversation with the client (spoofing the decoy).

forge_socket's role is hard to depict; its position in Figure 5.2 is misleading. The forge_socket logic never touches any packets or payloads. Rather, forge_socket allows userspace code to overwrite IP addresses, ports, initial sequence numbers, etc. on what otherwise functions as an ordinary TCP socket — meaning that forge_socket's "unsafe" C code is mostly just overwriting numerical variables in Linux kernel `structs`. All TCP segmentation, retransmission, and congestion control logic is performed by the host kernel's TCP implementation.

We have measured the TapDance station's performance in inspecting traffic. Checking for a TapDance tag, with Elligator's expensive elliptic point arithmetic, costs on average 274μs, with standard deviation of 1.5μs (measured over 100,000 tag checks), on a Xeon E5-2643v4 CPU. (The stations' hardware was not uniform, but all were of roughly this ballpark of power).

Each flow is processed entirely independently of all others, making the station embarrassingly parallelizable, assuming we are sure that successive overt flows in a single TapDance session all reach the same core. In this implementation, we hashed flows onto cores based only on the IP source and destination, rather than the full TCP 4-tuple. This choice allows a TapDance session to span multiple TLS streams—necessary because TapDance is severely limited both in how much data the client can send in one TLS stream, and how long the stream can be held open, before the decoy's muting ends, and the decoy breaks the session—with each successive stream coming back to the one core that knows about the ongoing session. The need to keep a session on the same core across streams also requires the same decoy server to be used for the entire session.

The client was implemented in Go, and was designed as a library presenting a reliable transport interface. That is, it shares an API with Go's TCP and TLS implementations, and is interchangeable with them. Psiphon built a version of their client that uses this library, and distributed this version to a small slice of their users. The integration with Psiphon somewhat altered TapDance's behavior: the Psiphon client creates a single tunnel that all traffic

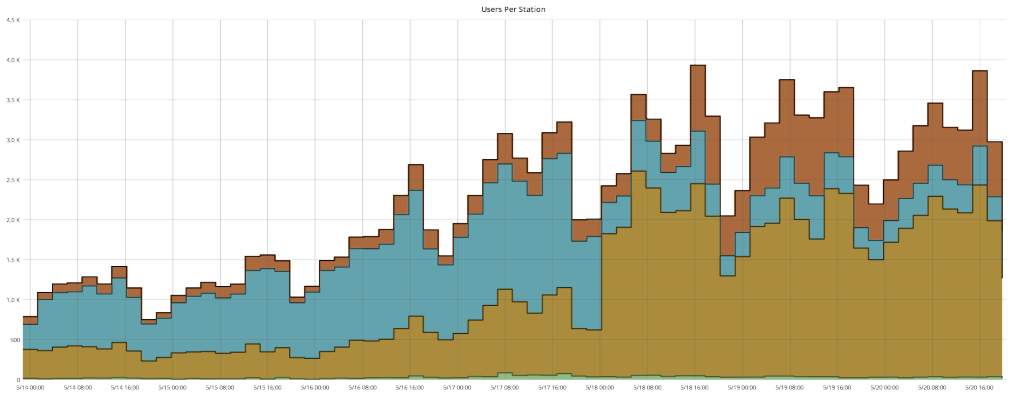

Figure 5.3: Concurrent active users at each of four stations. The curves are stacked not overlaid: the peak was around 4,000 total users.

flows through, meaning Psiphon-over-TapDance uses a single long-running TapDance session (with many reconnects). A stand-alone TapDance client, on the other hand, initiates a new TapDance session for every new TCP connection the browser initiates. The use of a single, long-lasting, high-usage tunnel makes the work to be described in chapter 6 crucial for future, larger-scale Psiphon-over-TapDance deployments.

5.3 Deployment

We deployed the system in 2017. All of our users were existing Psiphon users given a TapDance version of Psiphon through an upgrade. We deployed four TapDance stations: one tapping U. Colorado's 10 Gbps link to the internet, and three in Merit: one on a 10 Gbps link in Detroit, and one each on two different 40 Gbps links in Chicago. Additionally, we had a second station on the same Detroit link reserved for development and testing efforts. Across our four production machines, we had station processes running on a total of 34 cores.

The least powerful station (on one of the 10 Gbps links) had an Intel Xeon E5-2643v4 CPU and 32GiB of memory. The most powerful station (on one of the 40 Gbps links) had an 8-core Intel Xeon E5-2667v3 at 3.2GHz, 64 GiB of memory. The 40 Gbps links were tapped with quad-port Intel X710 10GbE SFP+ network adapters.

At the peak of the deployment, we had around 4,000 concurrent users,

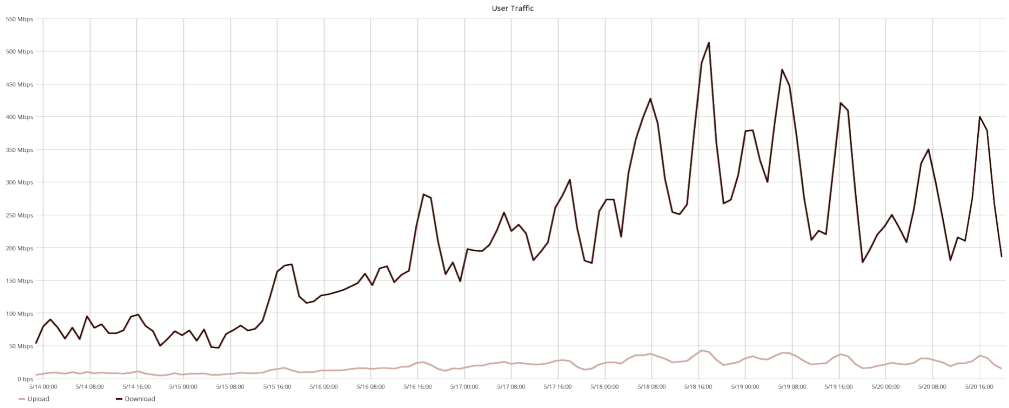

Figure 5.4: TapDance users' aggregate throughput.

getting an aggregate of up to 500 Mbps download throughput. We did not observe any attempts by the censor to block TapDance; not surprising given how new and short the deployment was. We did, however, see an unexplained surge of 1,000 sessions quickly ramp up, remain active for about 15 minutes, and then ramp back down, with transferring any significant data. We suspect this might have been some initial investigation by the censor into an unfamiliar protocol newly included in Psiphon. However, we did not observe any attempt by the censor to block TapDance—unsurprising, given the relatively short duration of the trial.

5.3.1 Decoy Selection

The moderate size of this deployment, with stations tapping just four links, required effort to provide clients with decoys that would take them on paths through stations. Furthermore, regardless of topological considerations, TapDance is slightly picky about its decoys. TapDance's muting of the decoy is somewhat fragile: a client uploading too much data or holding the HTTPS session open too long will cause the decoy to break the session. Additionally, only some symmetric ciphers are compatible with its chosen-ciphertext steganography (our implementation supports only the AES_128_GCM family of ciphersuites).

To give clients usable decoys, we ran an automated process to enumerate potential decoys (*i.e.* HTTPS servers), and test their compatibility as a TapDance decoy. The process scanned all IP addresses within the ASes of

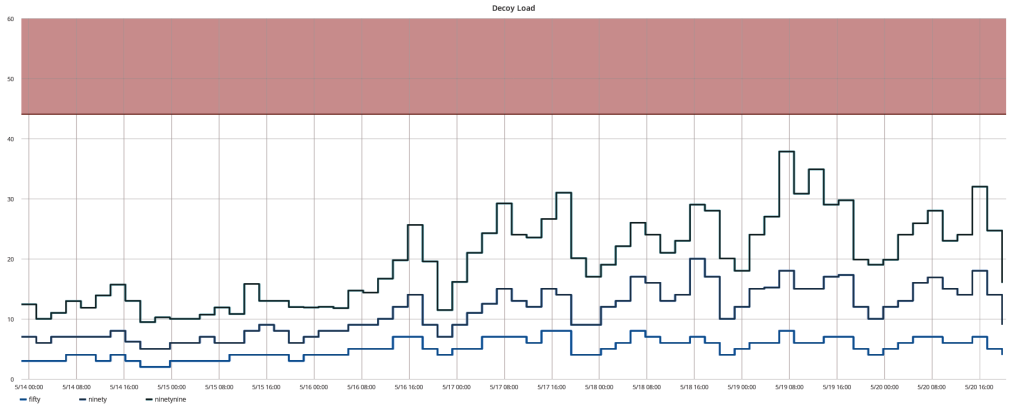

Figure 5.5: Decoy load: 50th, 90th, and 99th percentile. The red line is the system's current load limit: when any decoy reaches that number of active users, the system instructs all stations to stop accepting new users on that decoy.

U. Colorado, Merit, and Merit's customers, as users in foreign countries are highly likely to pass through a station on their way to any of these addresses. It measured advertised TCP receive window and wait time before the HTTP server would reject an incomplete HTTP request (the two factors that limit TapDance's muting). Finally, it used our TapDance client to attempt an actual TapDance session with that decoy, repeated three times.

From all decoys that consistently supported TapDance sessions, we removed those whose initial TCP receive windows were less than 15KB, and those whose HTTP timeouts were less than 30 seconds. We then randomly selected half (keeping the other half in reserve in case some contingency introduced a need to quickly distribute many more decoys) for the final list distributed to clients. Figure 5.6 shows how many decoys were active over time. By the time the trial was fully ramped up (around May 18th), essentially all usable decoys distributed to clients were in use. Keep in mind that the set of decoys distributed to users, and so represented in Figure 5.6, contained only half of all usable decoys discovered by the automated process.

The TapDance-enabled Psiphon app was bundled with our initial version of this list. Once the trial began, we ran this process daily to catch churn in active potential decoy servers in the relevant address space. We distributed the updated lists to clients by sending them within a TapDance session, whenever a client with a stale list version established a session.

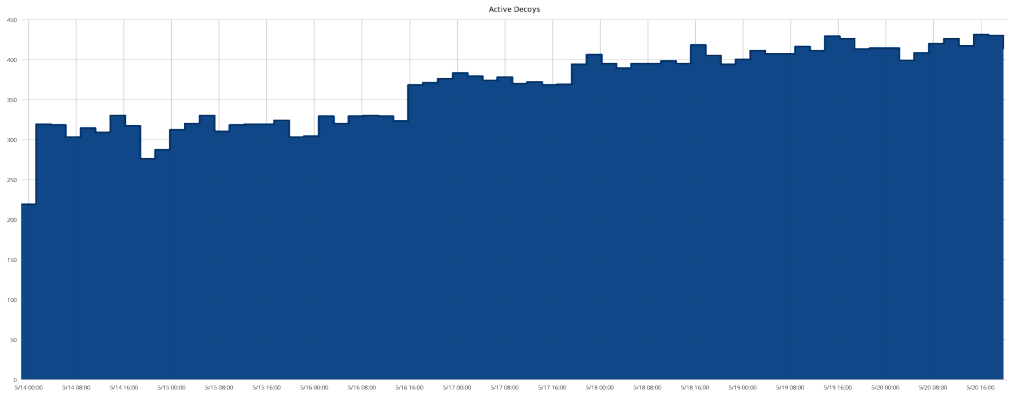

Figure 5.6: Number of decoys with at least one TapDance client currently connected.

5.3.2 Politeness to Decoys

One extremely important aspect of this or any other decoy routing deployment is politeness to decoys. Previous decoy routing work—strictly design proposals and small-scale prototypes—have all taken the concept of a decoy server for granted. However, when a decoy routing deployment is not so extensive that practically every HTTPS server on the internet is a viable decoy, a deployment seeking to serve many users must take care to avoid effectively (if not intentionally) DDoSing the decoys. Although the muting process prevents TapDance from wasting much of the decoys' bandwidth, HTTP servers do limit concurrent active connections. With enough TapDance users on one decoy, legitimate visitors could be shut out. It turns out that the popular Apache server defaults to a quite small limit of 150 concurrent clients. Although a site seeing significant traffic would be configured to something other than the defaults, many of our decoys are individual people's small-time servers, and so we must respect this conservative limit.

To enable the system to avoid overloading decoys, we built a central monitoring component. Because different users can pass through different stations to reach the same decoy, a global view of decoy usage is necessary. All stations keep this central tracker updated with their current decoy usage counts. The tracker sums these counts together, and if any rises to the "overload" threshold (we used 30, and later 45, in our deployment) it will instruct all stations to turn away new clients. Once the overload condition passes, the tracker informs the stations that the decoy is usable again.

Not DDoSing random bystanders is the baseline for responsibility. It is

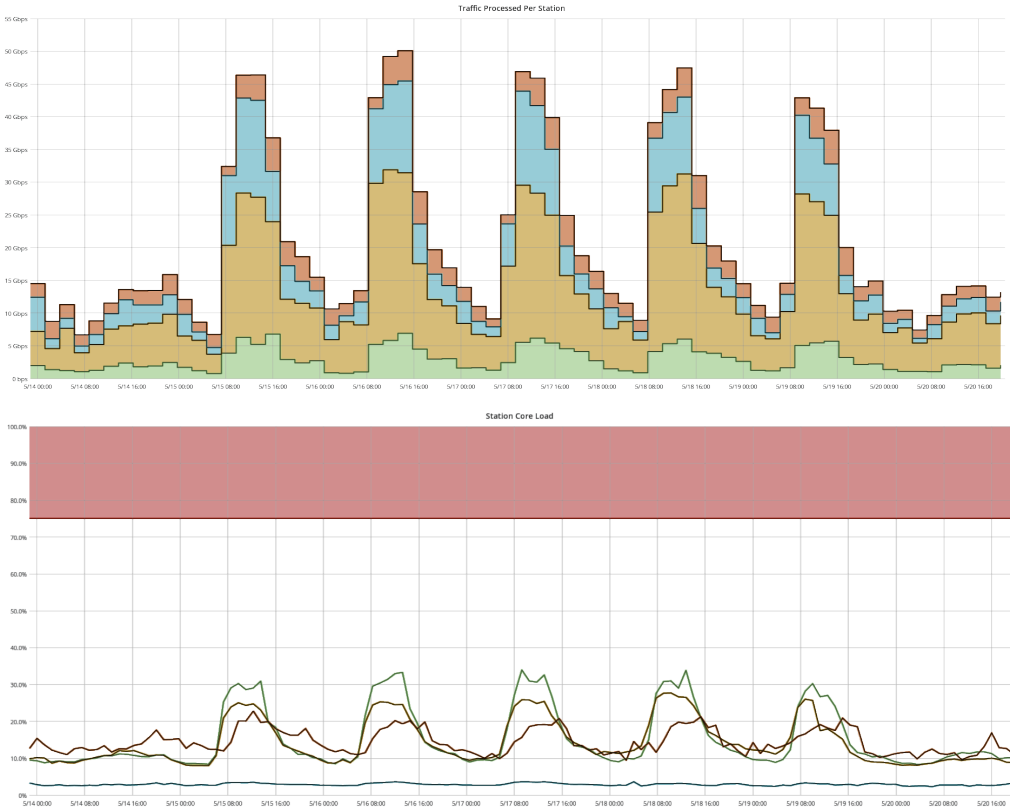

Figure 5.7: Background traffic processed and CPU load, at each of four stations. Essentially all of TapDance's CPU time is spent inspecting the entirety of the traffic seen on the link for TapDance sessions. Of that time, the majority is spent attempting to decode Elligator code points.

also desirable to allow administrators of potential decoys to opt their servers out of being decoys. To that end, our special incomplete muting-causing headers have their User-Agent field set to TapDance, along with a link: `https://tapdance.team`. That site contains a short description of TapDance and its relation to decoys, and instructions on how to use the server's `robots.txt` search engine preferences file to instruct our automated decoy harvesting process to exclude the server. Throughout the deployment, over 400 decoy servers were in essentially constant use by thousands of TapDance clients (see Figure 5.6). None of them used the `robots.txt` opt-out or contacted us, so TapDance appears not to be destructive to decoys in practice.

5.4 Challenges

We faced several challenges during implementation and deployment. First, the internet connection on one of our 40 Gbps stations—the one that happened to be the most popular with users—turned out to be 100 Mbps, rather than 1 Gbps. This severely limited the throughput available to users of our system. Because we did not have immediate physical access to these servers to change the connection, we instead had Merit reroute much of this station's overall traffic to the other 40 Gbps station. After the shift was made, and the other 40 Gbps station was handling the majority of TapDance user traffic, we began observing it having trouble keeping up with packet inspection: it was consistently having to ignore a small portion ($< 1\%$) to keep up. However, CPU usage was still at what appeared to be perfectly healthy levels. We concluded that the issue was likely that the time shares given to user traffic forwarding and packet inspection were too uneven, with so much time spent in traffic forwarding that the queues of packets to be inspected were overflowing. A further investigation of time share glitch will begin once the trial is complete.

In addition to the packet drops, this large traffic shift affected the pattern of load on decoy servers. In particular, 99th%ile load immediately hit our 30-concurrent-client cap, and consistently stayed there. Decoys in the overload state lead to additional load on the decoys: when a client attempts to connect to an overloaded decoy, it is turned away by the station, but only after initiating an HTTPS session with the decoy. If the next decoy it tries it is not overloaded, it has hit two decoys in order to start one TapDance session. If the next decoy *is* overloaded, it will have to move on to a third. This behavior causes decoy load to rapidly increase in the case where most decoys have reached the overload state. To avoid runaway behavior, the station is able to instruct the client to observe a total timeout from using TapDance, which can be used to implement exponential backoff.

To avoid reaching the runaway decoy load scenario, we sought to fix the overload condition immediately. Our options were to release additional decoys, or to accept a higher cap. As we felt our cap was very conservative, we fixed the problem by raising it to 45.

We also found that some of the decoys that our automated decoy collection process was collecting were, themselves, blocked in some censored

countries! In particular, Google's ubiquitous video CDN servers are often blocked. We manually removed these entries from the list. In future deployments, we would ideally have a host within countries of interest that would check whether decoys are available.

Finally, we observed some occasional strange behavior that we chalked up to multiple stations responding the same user. Multiple responses both break the session and are an obvious camouflage failure. Because stations respond immediately, and do not have time to consult each other, there is no way in the moment for a station to know whether any other station is also responding. Therefore, we must avoid the problem entirely.

When two stations are seeing the same request, they are on the same path to a decoy D. Call the station earlier on the path A. If a TapDance client connected to the router on the receiving end of the tapped link (call it C) were to use D, then the later station B would definitely see the request. If C cannot get a response, then there is no other station between A and D. If each station A declines to respond to TapDance sessions using any decoy D where C got a response from some B when using D, then for all paths from any real client to any decoy, only the last station on the path will respond. (We don't lose any station responses, though: any path that has one or more stations will still definitely have the last station respond). With an offline background processes constantly testing each newly discovered decoy in this way, we can avoid multiple responses.

CHAPTER 6

FASTER UPLOAD IN PASSIVE DECOY ROUTING

TapDance's key feature—the muting of decoy servers to allow the station to be a passive tap—places a tight constraint on TapDance users' upload throughput. The station spoofs the decoy when sending data to the client, which means the client ACKs data the decoy never sent. Data packets that the client sends therefore appear to the decoy to have erroneous ACK fields, and so the decoy ignores them. In particular, it does not advance its receive window as it would upon correct contiguous receipt. While the decoy will ignore a too-high ACK field, it will respond with a RST to data packets with a sequence field beyond its current receive window. Therefore, the client can only upload as much data in a TapDance TLS/TCP stream as the decoy's receive window at the time that muting took effect.

When the client hits the upload limit, it must continue the TapDance session in a new TLS stream, incurring three round trips' worth of downtime during the TCP and TLS handshakes. Given how small TCP receive windows typically are early in a TCP flow (in our deployment, we assumed no more than 15KB was safe), this leads to extremely low upload throughput. Even with larger receive windows, TCP congestion control logic constantly being reset to the beginning of slow start would harm throughput.

In a TapDance session with heavy upload, download throughput is also hurt: the congestion control logic resets and TCP+TLS handshake downtime hurts the download direction just as much. Furthermore, a continuous series of two or three HTTPS connections per second, especially all to the same server[1], is not good for camouflage. Practically speaking, these issues limit the original TapDance design to simple consumption of web pages: any sort

[1]The client must stay on the same server for an entire session because it has no way (short of brute force search) to pick a new decoy that is guaranteed to lead it through the same station. We have discussed a potential new architecture, where stations merely identify TapDance traffic and then forward it to a centralized single server. Such an architecture would add overhead and complexity, but might be worth it.

of file upload would be very painful, and video chat would be impossible. All of these problems are magnified if a single TapDance session is used to multiplex all of a device's internet traffic—which is the case with the Psiphon app.

Fortunately, all of these problems can be solved.

6.1 Upload-only Streams

A TapDance session with near-zero upload can achieve nearly the full download throughput of the client's internet connection. Therefore, if a TapDance session's upload is interfering with its download, splitting the two into parallel TapDance sessions will unburden the download direction. The upload direction's original problems remain, but the fact that it no longer handles any download opens up a new opportunity. The upload problem is a result of TapDance's muting-related constraints, and TapDance's need to mute is a result of needing to spoof the decoy to send data to the client. If there is no data to send, spoofing and therefore muting is unnecessary. Without the constraints of muting, the client is free to reach its full upload throughput, assuming the decoy will accept large amounts of data without breaking the connection. Fortunately, it appears that most servers will allow an HTTP POST to a non-existent URL to go through before returning an error.

TapDance can employ upload-only streams by leaking the TLS key material with chosen-ciphertext steganography, as before, but *not* muting the decoy server: rather than an incomplete HTTP request, a well-formed POST request is sent. The station will not attempt to spoof the decoy, but rather simply passively eavesdrop on the TLS session. All ACKing of data is handled by the decoy, with the station guaranteed to have received anything the decoy has ACKed[2]. The decoy can send significant chunks of data—hundreds of kilobytes—in a single HTTPS flow, and from the outside, that HTTPS flow will look like a legitimate image upload (although if the client does not upload non-stop, the traffic pattern will look suspect).

In testing the current TapDance implementation's upload speed, we found that a client with a 2 Mbps upload bandwidth takes 30.964s, with a standard deviation of 0.795s, to send 1MB through TapDance. Additionally, it must

[2]So long as the station hardware is fully capable of line-rate processing of the tap traffic.

reconnect around 90 times. With a single HTTPS POST flow, the same data can be uploaded in 4.515s, with a standard deviation of 0.138s.

6.2 Implementation

The idea of splitting a TapDance session in two, with one stream being a passive upload, runs into a few additional challenges when actually implementing it.

First, the station handles upload-only TapDance streams very different from vanilla bidirectional TapDance streams; any attempt to respond to an upload-only stream would break the session. The station could examine the HTTP method (GET vs POST), but we chose to use one bit of an existing 'flags' field in our implementation's steganographic payload.

Second, maintaining a coherent session that straddles two concurrent flows is not trivial. To keep the communication orderly and reliable, a client choosing to use the split stream approach first establishes a traditional TapDance session, with a single bidirectional stream. Once the session is established, it sends a control message informing the station that it is yielding the ability to upload data in the bidirectional stream, and that the station should expect a passive upload-only stream to arrive. The upload-only stream confirms that it is acquiring uploader status from the specific yield message sent in the bidirectional stream. After this point, both the uploader and downloader can (independently) reconnect. The uploader may also yield, making the downloader back into a vanilla bidirectional TapDance stream. A stream marked "download-only" is only prohibited from sending application data; it may send control messages to manage its own state (such as informing the station of a need to reconnect due to an impending HTTP timeout).

Finally, the station is no longer maintaining a TCP endpoint for spoofing the decoy. If the decoy properly ACKs all data that the client sends, then the station is implicitly guaranteed to have seen all the data, but it is not likely to have arrived in order, due to TCP retransmissions. The passive eavesdropping logic must therefore include a TCP receive buffer. Fortunately, TCP flow control guarantees that this buffer will not need to grow any larger than the receive window advertised by the decoy.

This feature was not fully implemented and tested in time for the 2017

trial, but should be part of any future deployment. It will certainly be useful in the context of Psiphon: a Psiphon+TapDance client used for standard web browsing reconnects two or three times a second during active usage, due entirely to the upload cap issue.

CHAPTER 7

DITTOTAP: DECOY ROUTING WITH BETTER CAMOUFLAGE

TapDance [10] allows all of the host ISP's traffic to proceed untouched. Slitheen [23] perfectly mimics the overt site. An ideal decoy routing technique would combine the two. Such a design is possible. We have developed a new decoy routing technique along these lines, called DittoTap[1].

In TapDance, the decoy router sends covert data to the client in packets spoofed as coming from the overt server, but using its own TCP logic, and placing no restrictions on flow length. DittoTap makes its own, duplicate request for the same overt resource, and sends the client shells of the packets it receives from the duplicate connection, with payloads replaced with covert data. Sending exactly the packets that the real overt server sent guarantees an identical traffic pattern, and identical TCP and TLS options and behavior.

The traffic pattern must match not just for individual objects, but for the entire sequence of HTTP requests that retrieve the contents of a given web page. Slitheen runs a web browser parser to generate the appropriate follow-up requests from a root HTML document. DittoTap functions similarly, but with some additional problems to solve (described in section 7.5) due to its use of a passive tap.

7.1 Implementation

Like the deployable TapDance station, the DittoTap station is largely written in Rust. In fact, the DittoTap station effectively *is* the TapDance station, just with some extra logic inserted at the beginning of packet inspection, and with the TLS-to-client object from the TapDance station (`BufferableSSL`) replaced with DittoTap logic: `BufferableDittotap`. These plug-in addi-

[1]Slitheen was named for a shapeshifting alien from Doctor Who. DittoTap is named both for the shapeshifting Pokémon Ditto, and for the fact that it duplicates the client's decoy flows.

tions are illustrated in Figure 7.1. However, while `BufferableDittotap` has the same simple interface as `BufferableSSL`, it contains quite a bit of logic: it manually initiates (packet by packet) the duplicate HTTPS session, it passively eavesdrops on the client's connection to the decoy (including TCP receive buffer logic to handle out of order packets), and handles the combining of overt headers with covert data. These tasks entail maintaining, for each DittoTap session, multiple mappings from one TCP sequence space to another.

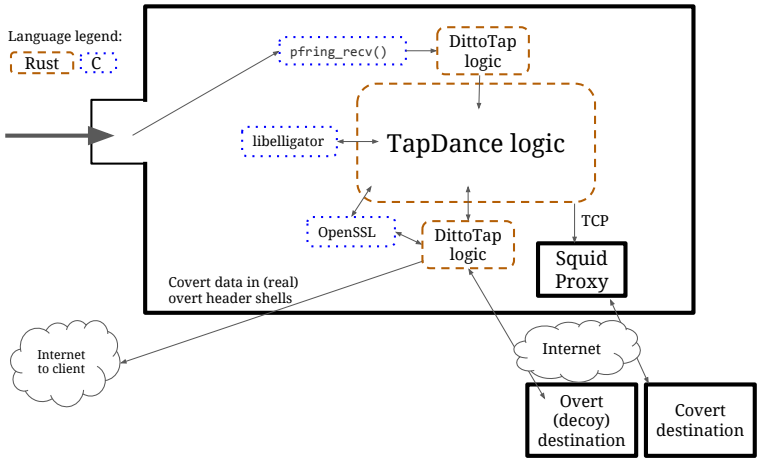

Figure 7.1: The architecture of the DittoTap station implementation, which was leaves essentially all of the TapDance station intact, and adds extra logic in initial packet processing, and in communication with the client.

Because the DittoTap station is built on top of the TapDance station, the stability, robustness, and performance characteristics remain the same. In fact, DittoTap eliminates one of the uglier parts of the TapDance station: the `forge_socket` kernel module. TapDance needs to spoof the decoy's TCP connection back to the client, but DittoTap has access to every actual TCP/IP header that the decoy sends (including retransmissions). Therefore, in DittoTap, the `forge_socket`-reliant TCP spoofing is replaced by the simple sending of raw IP packets, containing the duplicate overt connection's TCP headers.

7.2 TCP Traffic Patterns

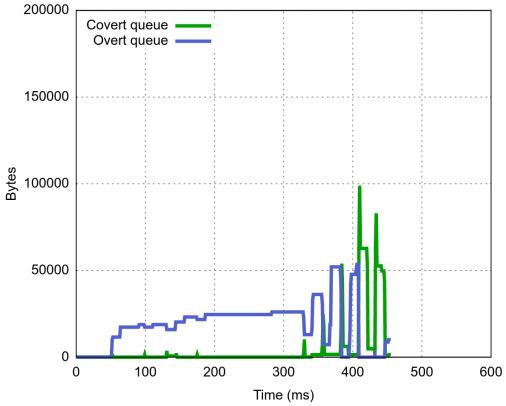

Figure 7.2: Queuing behavior when overt headers are queued as long as needed, for a download of the Twitter index page.

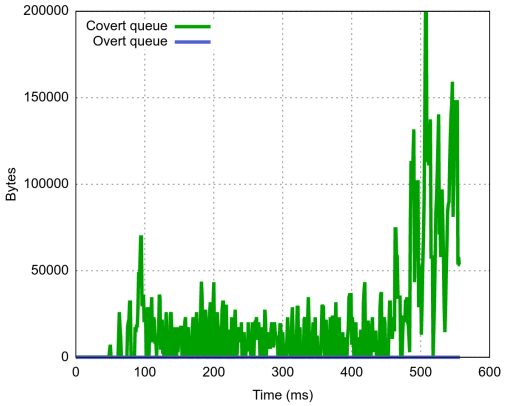

Figure 7.3: Queuing behavior when overt headers are always sent immediately, for a download of the Twitter index page.

Slitheen and DittoTap both have the goal of perfectly mimicking the decoy, down to the characteristics of the specific web object downloaded. A given HTTP server's TCP implementation, sending an object of a particular size under stable network conditions, will send packets at certain times, modified chiefly by congestion events. If the station inserts large delays among the packets it sends to the client, the censor has a chance to realize that the flow does not look like other download flows for that decoy. However, overt headers may arrive on the duplicate connection when there is not covert data ready to be sent. When this happens, we must either break our decoy traffic pattern imitation by queuing the overt header until covert data is available, or else send the overt header off with junk padding instead of real covert data. If many overt headers arrive when there is no covert data available, much of the client's download throughput will be wasted.

Figure 7.3 and Figure 7.5 show the covert and overt queues over time of ∼310KB downloads from Twitter and Imgur, respectively. In these experiments, this proper traffic-pattern-mimicking logic is followed; the majority of the covert queue is contributed by junk padding. (The overt queue is always 0, that being the logic's goal). This padding is highly wasteful; for the Twitter downloads, around 90% of the data sent to the client was wasted. Slitheen, or any other technique seeking to send one flow's data in another's traffic pattern with both flow happening in real time, suffers the exact same

amount of inefficiency.

Figure 7.3 and Figure 7.5 show the queue behavior of Twitter and Imgur downloads when padding is only added to fill out an overt header already partially filled with real covert data. This is the minimum padding possible; without it, the application logic communicating over DittoTap could stall. The overt queue stays non-zero for long stretches, meaning that a censor aware of the correct traffic pattern will notice delayed packets.

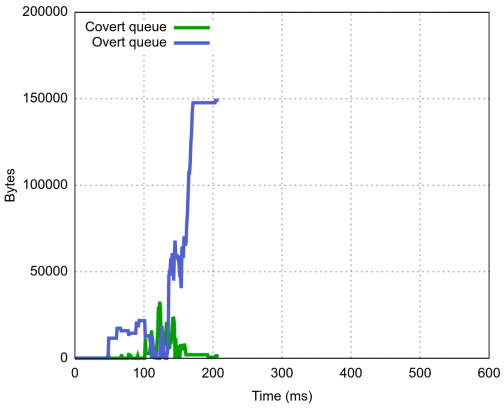

Figure 7.4: Queuing behavior when overt headers are queued as long as needed, for a download of an image (of the same size as the Twitter index page) from Imgur.

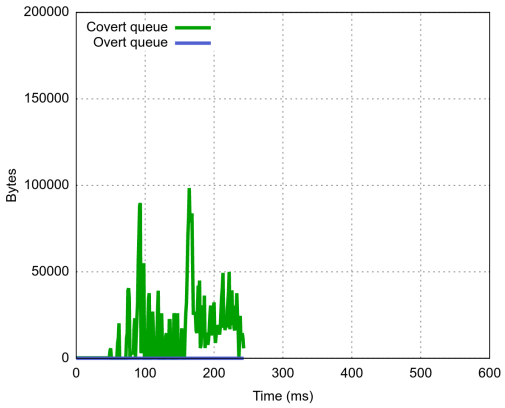

Figure 7.5: Queuing behavior when overt headers are always sent immediately, for a download of an image (of the same size as the Twitter index page) from Imgur.

There is a tradeoff available, between efficiency and the goal of perfect mimicking. If, rather than sending out overt headers immediately, even if there is no covert data to include in them, we instead allow them to wait for at least a little while, we give them an additional chance to receive covert data: see Figure 7.7 and Figure 7.8. The further apart the overt and covert traffic patterns are, the worse efficiency will be; equal sized downloads from Twitter perform worse than from Imgur. Furthermore, the overt header queue delay tradeoff appears to be much more useful for Twitter; Imgur already performs well.

Allowing a small amount of queuing helps efficiency, but does not greatly harm mimicry. Figure 7.6 shows the queuing behavior when downloading from Twitter with each overt header allowed to queue for up to 4ms. Although the overt queue becomes non-zero several times, it empties quickly—no more than 4ms. These small delays should blend in with natural variance in the traffic pattern.

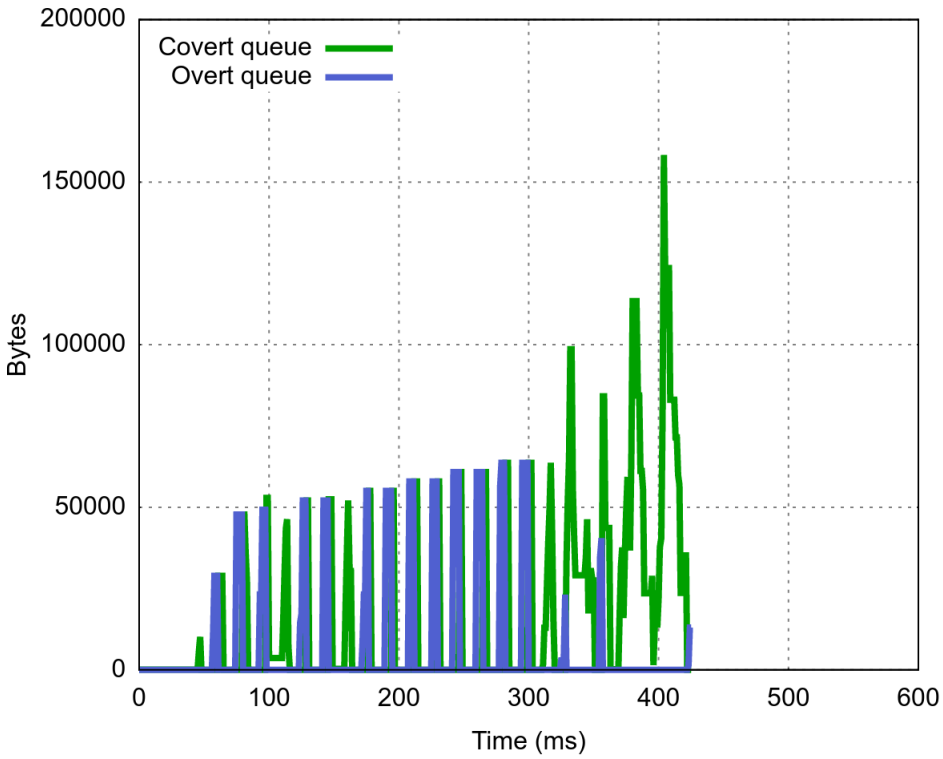

Figure 7.6: Queuing behavior when overt headers may be queued up to 4ms, Twitter.

Another possible solution (for pure web downloads) is to inform the station of the requested download well ahead (100s of ms) of the actual decoy routing connection that is to deliver the covert data to the user. This could be as simple as using two DittoTap sessions for one covert page retrieval: the first to convey the request (and perhaps also receive the page's root HTML document, in order to know what other objects must also be requested from the covert server), and the second to receive the response. This approach would establish a new tradeoff: a user who wants perfect traffic pattern mimicking can choose to either waste huge amounts of data on padding (undesirable on low bandwidth connections, or when being charged by the byte such as on a mobile device), or to experience delays in page loads. However, to make the most of the efficiency gains, DittoTap would have to maintain knowledge of the sizes of decoy pages: if the DittoTap session carrying the initial request does not use a very small decoy page, then that session itself contributes a large amount of waste.

Finally, there is room for the prototype implementation to optimize this issue of mismatched timing between the covert and overt downloads. In the

current TapDance/DittoTap implementation, the client sends no application data out-of-band. This means that the station cannot learn which covert destination to initiate a connection to until the TapDance/DittoTap session is fully underway. Therefore, the duplicate overt flow—which must be started ASAP to mimic the original overt flow—starts significantly before the covert flow, leading to many overt headers that have no chance to find covert data waiting for them. If the client could signal the station to begin the covert flow earlier, perhaps by including the destination and an initial chunk of data to send in the steganographic payload, the wait would be reduced.

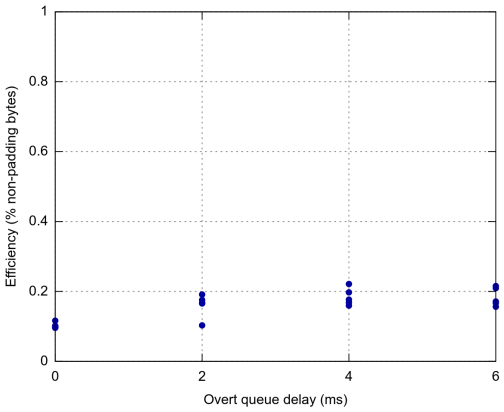

Figure 7.7: Efficiency (in terms of bytes wasted by padding) for different max overt queue waits, for downloads of the Twitter home page.

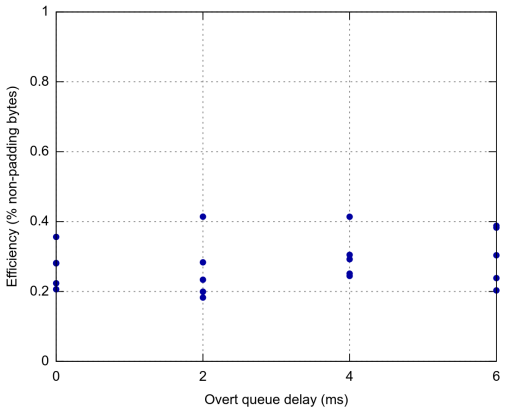

Figure 7.8: Efficiency (in terms of bytes wasted by padding) for different max overt queue waits, for downloads of an image from Imgur of about the size as the Twitter home page.

7.3 Page Load Patterns

The current DittoTap client is based on its TapDance counterpart even more cleanly than the station: specifically, there are no changes whatsoever! To make the overt traffic pattern fully realistic, the client would need to adopt Slitheen's realistic random overt browsing. Slitheen has clients randomly choose decoy pages to retrieve. This behavior is purely within the client, and unrelated to the specifics of any particular decoy routing design.

However, a major component of the realistic behavior is that the client receives an actual HTML document from the decoy, and then requests em-

bedded resources accordingly, with decoy routing only being performed on "leaf" objects such as images. Slitheen's approach, where the station decides on the fly whether each HTTP request is a leaf to be decoy routed, possibly switching back and forth, is not possible in DittoTap, whose required muting cannot be turned back off once it has begun. Fortunately, the DittoTap system does have some control: the client can allow a Keep-Alive HTTPS session (whose key material it has leaked to the station) to proceed unmuted as long as it wants before initiating muting with an incomplete header. Therefore, a client could actually download all non-leaf objects in a decoy page from the decoy, initiate muting, and then begin DittoTap with the leaf objects. In particular, a DittoTap client could load a simple HTML page embedding only images in this way, and be guaranteed to be behaving identically to any real browser.

For our initial proof-of-concept prototype, we consider implementing the full client behavior—which is conceptually relatively simple, and whose Slitheen counterpart has already been demonstrated to accomplish its camouflage goal—out of scope.

7.4 Avoiding Startup Delay

The DittoTap approach faces one challenge that Slitheen, with its inline, packet-blocking design, avoids. The duplicate overt connection should be identical to the original in every way. Packet timing is a crucial component, and if the system consistently inserts a pause anywhere, a careful censor could notice it. If the decoy router does not send its SYN right when it sees the client's SYN, it may end up inserting such a pause.

There is a simple if imperfect solution: using the same TLS ClientHello nonce tagging that Telex [21] used. As the ClientHello is the first packet sent in the TLS handshake, a decoy router that begins its duplicate overt connection in response to such a tag has only slept through the TCP handshake. In fact, if the station's RTT to the overt site is sufficiently lower than the client's, the duplicate connection can catch up, and eliminate the gap.

However, to guarantee there will be no gap without relying on station placement, the station needs to know whether a given flow will be duplicated when seeing the TCP SYN. Informing the decoy router ahead of time to

watch for a specific 4-tuple and TCP ISN would solve the problem, but we want simple, lightweight, user-friendly clients; that level of control over TCP requires root privileges. Instead, the client could notify the decoy router to watch for a future HTTPS connection, within a certain timeframe, from itself to a specific IP address.

Imagine that some host sharing the client's public IP address (perhaps the client itself), has started an ordinary TCP connection that fits a profile the station is expecting. The station will react by initiating a needless duplicate connection with the overt server. This unnecessary TCP initiation breaks nothing, however. Conversely, If the promised SYN does not arrive within the expected window, then the station will miss the chance to perfectly duplicate the connection, but can simply fall back to finding a Telex-style tag in the ClientHello. The size of the window within which the station duplicates a specified flow can therefore be used to control a tradeoff between false negatives (missing the SYN and relying on the ClientHello) and false positives (issuing spurious SYNs to the overt server).

We leave implementation of these techniques for future work, such as the development and large-scale deployment of a production-quality DittoTap system. The current prototype uses the same chosen-ciphertext steganography as TapDance, rather than Telex's ClientHello technique, or the advance-notice technique described in this section. The advance-notice technique would add significant complexity to the system, and should not be implemented until it is established that duplicate overt flows will in practice frequently be unacceptably delayed.

7.5 Full Page Mimicking

Slitheen mimicks not just the TCP/TLS behavior and packet timing of downloads from the decoy server, but also the pattern of requests of HTML objects from a real page, including the client's uploads that convey those requests. Mimicking both the packet size and timing of individual HTML object retrievals, along with the order in which those objects are retrieved, are the true strength of Slitheen. DittoTap should therefore achieve the same.

There are two options for driving the recursive retrieval of full HTML pages. A browser simulator that parses the containing HTML and makes

follow-up requests must be employed; that simulator can be run either on the station (in which case the client simply specifies the root document), or in the client (in which case the client makes a series of requests for specific objects). Either works for the purpose of displaying the correct download pattern, but the former would make it difficult to emulate the site's *upload* pattern: because the client would be unaware of the sequence of requests, it would not know the shape that its outgoing traffic ought to take.

Therefore, we prefer the design in which the client runs a browser's parsing and generates the individual HTTP requests itself. In this approach, the client sees the HTTP requests that its browser wants to make, which enables the client to send the correct pattern of packet sizes and timings. Just as in Slitheen, the decoy's pages will include leaf and non-leaf resources. To complete a full recursive page load, the client must see the actual content of the decoy's non-leaf resources (including the root document), in order to continue down the tree. Unlike in Slitheen, it is not possible to allow the decoy to directly respond to the non-leaf requests in the original decoy connection. Therefore, the station's covert channel back to the client must carry both the decoy's actual non-leaf data, as well as data from the covert destination.

Due to the overt non-leaf data and covert data sharing a channel, some multiplexing mechanism is necessary. Even without considering implementation details, the multiplexing will naturally introduce non-zero overhead. This overhead is a problem in this context: the station must pass overt packets along to the client as soon as they arrive and without inflating their sizes, and some of these packets (the non-leaf overt) must convey their original data. In particular, the first data the client receives is the contents of the decoy page's root HTML document, which the client needs in order to carry out the session. If the non-leaf contents are sent completely unaltered, their packets will have no space for our multiplexing overhead.

There are a few options for solving this problem:

- Keep the overhead entirely within the leaves. Accomplish the multiplexing with a state-transition approach, where the station sends tokens indicating "now sending covert data" and "now sending overt non-leaf data." All sessions begin in the "overt non-leaf" state, and transition tokens are considered covert data.

- Drop the HTTP response header. It occupies overt space, but the client does not need to see it.
- Compress the non-leaf page data, with the savings hopefully covering the multiplexing overhead.
- Lossy compression: make substitutions in non-leaf HTML guaranteed not to affect subsequent retrievals, such as removing comments and replacing `<div class="yt-masthead...">` with `<div>`.

All four techniques are compatible with each other; the best approach may be to use them all. The first *guarantees* that non-leaf objects' data will be transmitted within their own packet husks, while the other three optimize download throughput by freeing up portions of non-leaf space for covert data. However, the state-transition approach introduces the possibility for the client and station to get out of sync on what state they believe the communication is in. Sending non-leaf data in standalone packages in between covert data, relying on the three space saving techniques, is a simpler approach. So long as the saved space is enough to compensate for the packaging's overhead, this simpler approach is acceptable. In our implementation's current packaging of the non-leaf data (using Google's Protocol Buffers inside of our own framing protocol), each non-leaf object incurs 7-9 bytes of overhead, depending on the size of the non-leaf object. Therefore, the required beginning of the HTTP response—"`HTTP/1.1 `"—is by itself enough to accommodate our overhead. The two compression techniques are then useful as throughput optimizations, but not necessary.

7.6 Summary

DittoTap combines the deployability of TapDance with the extreme camouflage of Slitheen. Although it has not seen the same sort of real-world trial in an ISP that proved TapDance's deployability, the fact that its prototype was built as an add-on to the TapDance station, without disrupting the packet processing architecture, implies that from an ISP's point of view, DittoTap is no different from TapDance. However, DittoTap (as well as Slitheen) puts noticeably more load on decoys than TapDance does, due to the decoy receiving full HTTP requests that it must serve objects in response to. Given

that decoy load was an area of concern in the TapDance deployment, this issue would need even more careful attention in a DittoTap deployment.

The most important thing to understand about DittoTap is that it, like Slitheen, is first and foremost a solution to the not-yet-seen problem of censorship via website traffic fingerprinting. In contrast to TapDance, the client will not attain their full download bandwidth, as much throughput must be wasted on padding to stay in sync with the overt traffic pattern. We have investigated optimizations, but the waste is not entirely avoidable. Furthermore, the fact that it is not possible to maintain an idle but ready DittoTap session (without constantly wasting data on padding) makes integrating with whole-device tunnels, like the Psiphon app, impractical. Therefore, real-world deployments should use TapDance for the time being. Only when it is established that a TapDance deployment is being blocked by traffic fingerprinting should DittoTap be deployed.

CHAPTER 8

CONCLUSION

This dissertation describes a set of solutions to the problem of internet censorship. The various techniques can be described in two dimensions: deployability vs. effectiveness, and addressing the problem of access censorship (the blocking of websites) vs. publication censorship (limits on people's speech on internet platforms). Most of the work (and most previous work, academic and otherwise) addresses access censorship.

Salmon, a proxy server distribution system, focuses on deployability; it is a technique that individual free citizens could immediately deploy, with incremental effect per additional volunteer. The decoy routing technique, in which censorship circumvention is built into the middle of ISPs' networks, is much harder for the censor to block or degrade than proxy networks (even with Salmon's intelligent distribution)—although proxy server networks are easier to establish. DittoTap and our high-performance TapDance implementation, especially with the upload throughput improvement, would be/are extremely powerful, assuming there are ISPs willing to deploy the systems at their routers. While this reliance on ISPs makes TapDance and DittoTap less deployable than Salmon, they are still much more deployable than the original, packet-blocking decoy routing schemes. Furthermore, the successful completion of a real-world trial running at routers in a real ISP demonstrates that TapDance truly is deployable.

Against publication censorship, GhostPost is a simple way for users to undo some of the censor's efforts on social media. Similar existing systems are centralized, preserving the posts of a set of users chosen ahead of time by the administrators, whereas GhostPost is focused on the accounts its users follow. Furthermore, the other systems are more focused on measuring censorship, whereas GhostPost is designed for transparent everyday usage, automatically inserting deleted posts back exactly where they belong on the social media platform.

8.1 Future Work

Both censors and their opponents continue to develop new techniques. In particular, traffic fingerprinting, the issue addressed by Slitheen and DittoTap, has not yet seen widespread deployment[1], but would be extremely powerful. Similarly, this work has directions it can evolve in. Salmon, the decoy routing projects (DittoTap and deployable TapDance), and GhostPost all have potential for improvements and additions, both in theoretical design and implementation.

When decoy routing is only deployed on a few servers, it will likely only have around a thousand viable decoys, at which point the censor could treat the decoys themselves as proxy servers and block them (since many decoys will be odd corners of the internet representing next to no collateral damage when blocked). Using Salmon to distribute the decoys would greatly mitigate that danger in such a scenario.

The TapDance deployment will hopefully eventually be repeated at a larger scale and permanently, and with the significant upgrade of upload throughput (and camouflage) described in chapter 6. DittoTap should be carefully implemented and deployed in the same scale of trial as the recent TapDance trial, to ascertain its actual real-world performance. It should provide noticeably worse throughput than TapDance, and so has no need to be deployed until censors are actually employing traffic fingerprint censorship, but it should be tested and ready to go.

GhostPost could be deployed alongside the automated homophone substitution of [35], which is a completely orthogonal approach. In that case, users' own posts would be somewhat better shielded by the homophone substitution, while posts of the users they follow would be protected by GhostPost. GhostPost always preserves a user's own posts, so automated homophone substitution bundled with GhostPost would only be useful to the followers of GhostPost users.

While the addition of automated homophone substitution would be a nice optimization, the most important thing for GhostPost is to gain a large userbase. While even the very GhostPost first user will gain some benefit, in that

[1] Other than an attack [3] against Tor, whose pattern was a much easier fingerprinting target than the more general problem of identifying particular web sites from their traffic pattern.

their own instance of GhostPost will sometimes preserve posts they otherwise would never have seen, GhostPost's effectiveness at preserving posts increases dramatically until about 0.1% of Weibo users are GhostPost users. Ideally, some group with a large audience of Weibo users would promote GhostPost, to get many users to join simultaneously, and thus make GhostPost much more effective, and therefore attractive to other prospective users. There is already some interest among the Chinese internet freedom community [**?**].

Finally, to ease deployability, the implementation of GhostPost should either come bundled with access-censorship-circumvention, or should communicate with the central server over an uncensorable channel such as encrypted email. Otherwise, once a GhostPost deployment grows large enough for the censor to take notice and block the central server, the userbase would be disrupted.

The most important of all possible future developments in censorship circumvention is the serious support of the government of a free country. Decoy routing is the most promising technique in terms of unblockability and performance, but it would require significant resources to deploy and maintain stations at the hundreds of routers that a true global-scale deployment would require. In the meantime, we can continue developing and testing the tools that such an effort would use.

8.2 Future Challenges

It is worth considering the political and practical realities forming the context of internet censorship. Consider the effect of truly perfect (unblockable, high performance, widely available, very easy to use) anti-censorship, where the government has no way to stop anyone from accessing any internet site or service. The censoring government still has the option of entirely cutting off internet access to the outside world. It may seem that in most countries, this move is not actually feasible: modern humans' lives are greatly tied up in the internet; such a move would be a disaster for the populace and, likely not long after, the government.

However, the idea that a censoring government would not practically be able to use the "shut down everything" option fundamentally assumes that much of the internet's utility for many of its users is its global reach. If a

"censored" internet user only regularly uses sites and services based within their country, they would not be particularly disrupted by a complete shutdown of internet access. If such internet users represented the majority of the country's users, a complete shutdown would be much more politically feasible.

China, the world's most significant internet censor, would seem to be in such a situation. For every internet site or service popular outside of China (*e.g.* Google, Amazon, Facebook, Twitter), there is a China-based equivalent (Baidu, Alibaba, Renren, Sina Weibo) that is much more popular within China. Not only are the non-Chinese services relatively less popular than the Chinese alternatives, they are also unpopular in absolute terms: *e.g.* only around 0.04% of Chinese are active Facebook users [47].

Therefore, it is not inconceivable that the Chinese government might react to perfect anti-censorship by largely cutting ties to the outside internet. Average individuals would still have access to the sites and services that make up the bulk of their internet use. Collateral damage could be further reduced by allowing access to the most popular foreign non-sensitive sites on a whitelist basis, or by restricting foreign access to internet cafes with tightly controlled software. In either approach, the censor would have to require that browsers speaking HTTPS with foreign sites trust the censor's root certificate (and not use certificate pinning), so that it could man-in-the-middle any session it chose (in order to fight decoy routing and domain fronting). Businesses could be issued with similarly monitored but unfiltered connections as necessary, to avoid economic disruption.

Against such censorship, anti-censorship's task would shift from finding ways to use an existing but censored internet connection to provide uncensored internet access, to outright providing the internet connection itself. The connection has to be provided from outside the censored country's physical territory, to a user on the inside. This could probably only be done with satellites. Assuming it would be difficult for the censored population to make (illegal) payments to a foreign provider, access would have to be free. Such a project would then have to be a serious undertaking by a major free country's government. Alternatively, the satellite internet service could make money by injecting ads. There is precedent for such a configuration: advertising targeting Iranians supports numerous Persian language satellite channels, which serve the (illegal) home satellite dishes that are ubiquitous

in Iran.

There are serious challenges ahead for the cause of internet freedom. Censorship may get worse before it gets better; existing censors could deploy new tools such as traffic fingerprinting, and censorship could spread if people allow their governments to be seduced by the idea of controlling ideas. However, the censors are fundamentally on the wrong side of reality: it is much easier to spread information than to contain it.

REFERENCES

[1] Neo, Sparks, Tank, Dozer, and Smith, "The collateral damage of internet censorship by DNS injection," in *SIGCOMM Computer Communication Review (CCR July 2012)*. ACM, 2012, pp. 21–27.

[2] D. Anderson, "Splinternet behind the Great Firewall of China," *Queue*, vol. 10, no. 11, p. 40, 2012.

[3] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website fingerprinting in onion routing based anonymization networks," in *10th annual WPES*. ACM, 2011, pp. 103–114.

[4] P. Winter and S. Lindskog, "How the Great Firewall of China is blocking Tor," *Free and Open Communications on the Internet*, 2012.

[5] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 1, pp. 5–16, 2007.

[6] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso et al., "Gt: picking up the truth from the ground for internet traffic," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 5, pp. 12–18, 2009.

[7] "Tor: Pluggable transports," https://www.torproject.org/docs/pluggable-transports.html.en, accessed: 2017-06-12.

[8] F. House, "Freedom on the net 2016," https://freedomhouse.org/report/freedom-net/freedom-net-2016, 2016.

[9] F. Douglas, W. Pan, M. Caesar et al., "Salmon: Robust proxy distribution for censorship circumvention," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 4–20, 2016.

[10] E. Wustrow, C. M. Swanson, and J. A. Halderman, "Tapdance: End-to-middle anticensorship without flow blocking," in *Proceedings of 23rd USENIX Security Symposium (USENIX Security 14). San Diego, CA: USENIX Association*, 2014.

[11] F. Douglas and M. Caesar, "Ghostpost: Seamless restoration of censored social media posts," in *6th USENIX Workshop on Free and Open Communications on the Internet (FOCI 16)*, 2016.

[12] D. Nobori and Y. Shinjo, "VPN Gate: A volunteer-organized public VPN relay system with blocking resistance for bypassing government censorship firewalls," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX, 2014. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/nobori pp. 229–241.

[13] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," DTIC, Tech. Rep., 2004.

[14] D. McCoy, J. A. Morales, and K. Levchenko, "Proximax: Fighting censorship with an adaptive system for distribution of open proxies," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, St Lucia, February 2011.

[15] Q. Wang, Z. Lin, N. Borisov, and N. Hopper, "rBridge: User reputation based tor bridge distribution with privacy preservation." in *NDSS*, 2013.

[16] N. Feamster, M. Balazinska, W. Wang, H. Balakrishnan, and D. Karger, "Thwarting web censorship with untrusted messenger discovery," in *Privacy Enhancing Technologies 2003*, Dresden, Germany, March 2003.

[17] R. Dingledine and N. Mathewson, "Design of a blocking-resistant anonymity system."

[18] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. Porras, "Evading censorship with browser-based proxies," in *Privacy Enhancing Technologies*. Springer, 2012, pp. 239–258.

[19] J. Karlin, D. Ellard, A. W. Jackson, C. E. Jones, G. Lauer, D. P. Mankins, and W. T. Strayer, "Decoy routing: Toward unblockable internet communication," in *USENIX workshop on free and open communications on the Internet*, 2011.

[20] A. Houmansadr, G. T. K. Nguyen, M. Caesar, and N. Borisov, "Cirripede: circumvention infrastructure using router redirection with plausible deniability," in *Proceedings of CCS*, 2011.

[21] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman, "Telex: Anticensorship in the network infrastructure," in *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[22] D. Ellard, C. Jones, V. Manfredi, W. T. Strayer, B. Thapa, M. Van Welie, and A. Jackson, "Rebound: Decoy routing on asymmetric routes via error messages," in *IEEE 40th Conference on Local Computer Networks (LCN)*, 2015, pp. 91–99.

[23] C. Bocovich and I. Goldberg, "Slitheen: Perfectly imitated decoy routing through traffic replacement," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1702–1714.

[24] A. Houmansadr, E. L. Wong, and V. Shmatikov, "No direction home: The true cost of routing around decoys," in *Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium*, 2014.

[25] M. Schuchard, J. Geddes, C. Thompson, and N. Hopper, "Routing around decoys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 85–96.

[26] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson, "Blocking-resistant communication through domain fronting," *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 2, pp. 1–19, 2015.

[27] A. Houmansadr, T. J. Riedl, N. Borisov, and A. C. Singer, "IP over Voice-over-IP for censorship circumvention," *CoRR*, vol. abs/1207.2683, 2012. [Online]. Available: http://arxiv.org/abs/1207.2683

[28] S. Li, M. Schliep, and N. Hopper, "Facet: Streaming over videoconferencing for censorship circumvention," in *Proceedings of the 13th Workshop on Privacy in the Electronic Society*. ACM, 2014, pp. 163–172.

[29] W. Zhou, A. Houmansadr, M. Caesar, and N. Borisov, "SWEET: Serving the web by exploiting email tunnels," *Privacy Enhancing Technologies Symposium*, 2013.

[30] J. Geddes, M. Schuchard, and N. Hopper, "Cover your acks: Pitfalls of covert channel censorship circumvention," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 361–372.

[31] T. Zhu, D. Phipps, A. Pridgen, J. R. Crandall, and D. S. Wallach, "The velocity of censorship: High-fidelity detection of microblog post deletions," *arXiv preprint arXiv:1303.0597*, 2013.

[32] K.-w. Fu, C.-h. Chan, and M. Chau, "Assessing censorship on microblogs in china: Discriminatory keyword analysis and the real-name registration policy," *Internet Computing, IEEE*, vol. 17, no. 3, pp. 42–50, 2013.

[33] F. Kw, M. Chau, and P. Holme, "Reality check for the chinese microblog space: a random sampling approach," *PloS one*, vol. 8, no. 3, p. e58356, 2013.

[34] GreatFire.org, https://en.greatfire.org/.

[35] C. Hiruncharoenvate, Z. Lin, and E. Gilbert, "Algorithmically bypassing censorship on Sina Weibo with nondeterministic homophone substitutions," in *9th Intl. AAAI Conf. on Web and Social Media*, 2015.

[36] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh, "Stegotorus: a camouflage proxy for the tor anonymity system," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 109–120.

[37] H. Mohajeri Moghaddam, B. Li, M. Derakhshani, and I. Goldberg, "Skypemorph: Protocol obfuscation for tor bridges," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 97–108.

[38] Q. Wang, X. Gong, G. T. Nguyen, A. Houmansadr, and N. Borisov, "Censorspoofer: asymmetric communication using ip spoofing for censorship-resistant web browsing," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 121–132.

[39] "Austrian Tor exit node operator found guilty as an accomplice," www.techdirt.com/articles/20140701/18013327753/tor-nodes-declared-illegal-austria.shtml, July 2014.

[40] "Tips for Running an Exit Node with Minimal Harassment," https://blog.torproject.org/blog/tips-running-exit-node-minimal-harassment.

[41] R. Dingledine, "Research problems: Ten ways to discover tor bridges," https://blog.torproject.org/blog/research-problems-ten-ways-discover-tor-bridges, October 31st 2011, accessed: 2017-06-12.

[42] Q. Wang and N. Borisov, "Octopus: A secure and anonymous dht lookup," in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*. IEEE, 2012, pp. 325–334.

[43] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing Privacy Enhancing Technologies*. Springer, 2001, pp. 46–66.

[44] P. L'Ecuyer, L. Meliani, and J. Vaucher, "SSJ: A framework for stochastic simulation in Java," in *Proceedings of the 2002 Winter Simulation Conference*. IEEE Press, 2002, http://simul.iro.umontreal.ca/ssj/indexe.html. pp. 234–242.

[45] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange, "Elligator: Elliptic-curve points indistinguishable from uniform random strings," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 967–980.

[46] S. Gallenmller, "Comparison of Memory Mapping Techniques for High-Speed Packet Processing," M.S. thesis, Technische Universitt Mnchen, 2014.

[47] GreatFire.org, "Facebook advertising platform's report of active users in china," https://twitter.com/GreatFireChina/status/250913333602029568, accessed: 2017-06-12.